NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DESIGN, IMPLEMENTATION, AND TESTING OF A REAL-TIME SOFTWARE SYSTEM FOR A QUATERNION-BASED ATTITUDE ESTIMATION FILTER

by

Ildeniz Duman

March 1999

Thesis Co-Advisors: Eric R. Bachmann
Robert B. McGhee

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE
DESIGN, IMPLEMENTATION, AND TESTING OF A REAL-TIME SOFTWARE SYSTEM FOR A QUATERNION-BASED ATTITUDE ESTIMATION FILTER

5. FUNDING NUMBERS

6. AUTHOR(S)
Duman, Ildeniz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Human limb segment angle tracking requires a system which can track through all orientations. The major problem addressed by this research was to develop a real time inertial motion tracking system based on quaternions to overcome the singularities of Euler angle filters.

This work involves clarification of the theory behind quaternion attitude estimation and development of a system capable of determining the orientation of an object in world coordinates. System sensors were built using miniature accelerometers, rate sensors, and magnetometers. The software system was designed by using Unified Modeling Language (UML) with object oriented design techniques. The actual implementation created a real time orientation tracking system.

The system was tested with dynamic tilt table experiments. Test results showed that the quaternion attitude estimation filter system can track human limb segments in real time within 1 degree of accuracy in any orientation and with a 55 Hz. update rate.

14. SUBJECT TERMS
Quaternions, Euler Angles, Quaternion Attitude Estimation Filter, Inertial Motion Tracking

15. NUMBER OF PAGES
181

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

DESIGN, IMPLEMENTATION, AND TESTING OF A REAL-TIME SOFTWARE SYSTEM FOR A QUATERNION-BASED ATTITUDE ESTIMATION FILTER

Ildeniz Duman
Lieutenant Junior Grade, Turkish Navy
B.S.C.S., Turkish Naval Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1999

Author: _____
Ildeniz Duman

Approved by: _____
Eric R. Bachmann, Thesis Co-Advisor

_____
Robert B. McGhee, Thesis Co-Advisor

_____
Dan Boger, Chairman
Department of Computer Science

ABSTRACT

Human limb segment angle tracking requires a system which can track through all orientations. The major problem addressed by this research was to develop a real time inertial motion tracking system based on quaternions to overcome the singularities of Euler angle filters.

This work involves clarification of the theory behind quaternion attitude estimation and development of a system capable of determining the orientation of an object in world coordinates. System sensors were built using miniature accelerometers, rate sensors, and magnetometers. The software system was designed by using Unified Modeling Language (UML) with object oriented design techniques. The actual implementation created a real time orientation tracking system.

The system was tested with dynamic tilt table experiments. Test results showed that the quaternion attitude estimation filter system can track human limb segments in real time within 1 degree of accuracy in any orientation and with a 55 Hz. update rate.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MOTIVATION 1

B. PROBLEM STATEMENT 1

C. ORGANIZATION 2

II. BACKGROUND 5

A. INTRODUCTION 5

B. REQUIREMENTS FOR SPATIAL TRACKERS 5

C. MOTION TRACKER PERFORMANCE 6

D. TYPES OF MOTION TRACKERS 8

1. Mechanical Tracking Devices 8

2. Electromagnetic Tracking Devices 10

3. Acoustic Tracking Devices 12

4. Optical Tracking Devices 13

5. Inertial Tracking Devices 15

E. RELATED WORK 16

F. SUMMARY 18

III. RIGID BODY ROTATION 19

A. INTRODUCTION 19

B. EULER METHOD 19

1. Rigid Body Rotation with Euler Angles 19

2. Body Rates to Euler Rates 22

3. Problems with Euler Angles 25

C. QUATERNION METHOD 26

1. Quaternion Basics 26
2. Rigid Body Rotation with Quaternions 30
3. Derivation of Quaternion Rates from Body Rates 31
4. Concatenating Rotations 33
5. Computing Rotation Matrices from Quaternions 33
6. Slerp 34
D. COMPARISON OF EULER ANGLES AND QUATERNIONS 35
E. SUMMARY 36
IV. QUATERNION ATTITUDE ESTIMATION FILTER 39
A. INTRODUCTION 39
B. QUATERNION ATTITUDE ESTIMATION FILTER 39
C. DERIVATION OF THE FILTER 42
1. Linearization 46
2. Response to Initial Condition Error 47
3. Rate Sensor Bias Correction 49
D. WHITE NOISE EFFECTS 51
E. SUMMARY 54
V. SYSTEM HARDWARE CONFIGURATION 55
A. INTRODUCTION 55
B. HARDWARE DESCRIPTION 56
1. Sensor Block 56
2. A/D Converter 59
3. Other Components 59
C. SUMMARY 60
VI. SOFTWARE DEVELOPMENT 61

A. INTRODUCTION 61
B. SYSTEM REQUIREMENTS ANALYSIS 61
C. SYSTEM ANALYSIS 66
D. SYSTEM DESIGN 69
E. SYSTEM IMPLEMENTATION 76
F. SUMMARY 77
VII. SYSTEM TESTING 79
A. INTRODUCTION 79
B. SYSTEM CALIBRATION 79
C. TESTING METHODOLOGY 80
D. TEST RESULTS 80
E. SUMMARY 82
VIII. SUMMARY AND CONCLUSIONS 85
A. SUMMARY 85
B. CONCLUSIONS 85
C. RECOMMENDATIONS FOR FUTURE WORK 86
APPENDIX A. DERIVATION OF X MATRIX 87
APPENDIX B. ..... DERIVATION OF GRADIENT OF THE ERROR CRITERION FUNCTION 91
APPENDIX C. SOURCE CODE 95
A. QAEF.H 95
B. QAEF.CPP 97
C. FILTER.H 99
D. FILTER.CPP 102
E. CONVERTER.H 111

F. CONVERTER.CPP 114
G. SAMPLER.H 122
H. SAMPLER.CPP 124
I. MATRIX.H 130
J. MATRIX.CPP 132
K. QUATERNION.H 139
L. QUATERNION.CPP 142
M. UTIL.H 151
N. UTIL.CPP 152
O. TEST.CPP 154
LIST OF REFERENCES 157
INITIAL DISTRIBUTION LIST 161

LIST OF FIGURES

Figure 1: Euler Angle Attitude Filter [HENA97] 17
Figure 2: Right Handed Earth Coordinate Axis 20
Figure 3: Body Coordinate Axes 22
Figure 4: Quaternion Attitude Estimation Filter 40
Figure 5: Upper Left Part of Quaternion Attitude Estimation Filter 42
Figure 6: Linearized System Isolating Error 46
Figure 7: SFG for Initial Condition Error Analysis 47
Figure 8: Transform Domain SFG for $\hat{x}(s)$ 48
Figure 9: Bias Estimation Filter 50
Figure 10: Lower Left Part of Quaternion Attitude Estimation Filter 51
Figure 11: Modified Filter with White Noise 52
Figure 12: 10 Degree Offset, a=0.1, ?t=0.1, No Noise 53
Figure 13: 10 Degree Offset, a=0.1, ?t=0.1, With Gaussian White Noise, Noise Standard Deviation=0.57 Degree 53
Figure 14: System Hardware Configuration 55
Figure 15: The Sensor Block [MCKI98] 56
Figure 16: Honeywell HMC2003 3-axis Magnetic Sensor Hybird [HONE98] 57
Figure 17: Crossbow CXL04M3 3-axis Accelerometer [CROS98] 57
Figure 18: National Ins. PCI-MIO-16XE-50 Data Acquisition Card [NATI98] 59
Figure 19: National Ins. SCB68 I/O Connection Board [NATI98] 60
Figure 20: Use Case Diagram 62
Figure 21: Context Diagram 62
Figure 22: Sequence Diagram 63
Figure 23: Object Diagram 67
Figure 24: System Statechart 67
Figure 25: Class Diagram 68
Figure 26: System Deployment Diagram 70
Figure 27: Filtering Subsystem Structure 71
Figure 28: Filtering Subsystem Operations 72
Figure 29: Filter Object State Model 73
Figure 30: Sampler Object State Model 73
Figure 31: Acquisition Subsystem Structure 74
Figure 32: Acquisition Subsystem Operations 75
Figure 33: Converter Object State Model 75
Figure 34: 45 Degree Roll Test, 10 deg/sec, a = 0.054, 55 Hz. 81
Figure 35: 45 Degree Pitch Test, 10 deg/sec, a = 0.054, 55 Hz. 81
Figure 36: 45 Degree Yaw Test, 10 deg/sec, a = 0.054, 55 Hz. 82

LIST OF TABLES

Table 1: 3-axis Magnetometer Specifications 57
Table 2: 3-axis Accelerometer Specifications 58
Table 3: 3-axis Angular Rate Sensor Specifications 58
Table 4: Sensor Channel Connections 59
Table 5: System Goals 64
Table 6: System Use Cases 64
Table 7: System Functions 65

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to Dr. McGhee and Eric Bachmann for their help during this research. Dr. McGhee is a fine teacher, and full of passion for teaching. He is always ready to give something from his deep knowledge. His experiences and his smart ideas allow a different view for everything. Beyond these he has an incredible patience that made me sometimes feel guilty. Eric Bachmann provided me the historical background of this research. He was always ready for questions and never hesitated to answer. He is very good at discussing the problems and pointing out the solutions. I would like to thank him for his friendly assistance. I could not have had two better advisors, and it was a pleasure for me working with you.

DEDICATION

For the memory of Alper Tunga AKAN

I. INTRODUCTION

A. MOTIVATION

For many years, computer scientists have tried to create realistic virtual environments (VE) for simulation, training, evaluation, data visualization, computer aided design, tele-operation, tele-presence, robotics and entertainment. During this period, computer performance increased rapidly in both hardware and software [HENA97]. Many techniques and devices have been discovered and introduced to interface computers and humans. These advances are needed to deal with the well-known "human machine interaction problem" [HENA97]. Special interaction devices are needed to achieve the goal of total immersion of humans into virtual environments [HENA97]. The main thrust of research in this area has been directed toward producing new and improved sensors for tracking objects and humans.

There are currently five fundamental tracking technologies; namely, mechanical, electromagnetic, acoustic, optical and inertial [DURL95]. Very recently, due to advances in component technology, inertial tracking systems have become more promising than other tracking technologies [SKOP96]. Inertial tracking systems use a combination of angular rate, accelerometer and magnetic sensors. Filter algorithms combine these sensor readings to obtain limb segment orientations. Such trackers are free of most of the problems of other tracking systems.

B. PROBLEM STATEMENT

Several methods exist for describing the orientation of an object. The most widely used method is to use Euler angles to describe attitude. However, tracking

systems using Euler angles are not capable of tracking objects in all orientations due to gimbal lock singularities [BACH96, SKOP96].

A quaternion based attitude estimation filter has been proposed to overcome the singularities and the divide by zero errors encountered when using Euler angles to represent orientations [MCGH96]. The theory behind the quaternion attitude estimation filter has been proved to be correct and a simulation program has been written in ANSI Common Lisp using only static sensor readings [HENA97]. The goal of this thesis is to develop a real-time quaternion-based orientation estimation filter in a PC environment, quantitatively test this system with a tilt table, and qualitatively test it when mounted on a human limb segment [USTA99]. Before the real-time software development, white noise effects on this filter are examined with a simulation program. In this thesis, the Unified Modeling Language (UML) [DOUG98] is used for software development, and the C++ programming language for system implementation.

C. ORGANIZATION

Chapter II of this thesis surveys existing tracking technology and related work in the area of inertial tracking systems. Chapter III discusses representations of rigid body orientation with Euler angles and quaternions, and presents a comparison between Euler angles and quaternions. Chapter IV presents the detailed mathematical theory behind the quaternion attitude estimation filter, white noise effects on the filter, and the derivation of the gain constant in the filtering process. Chapter V explains the hardware configuration of the system. Chapter VI presents the software design and development process. Chapter VII presents the test results of this real time implementation of the filter with a tilt table. The last chapter, Chapter VIII, presents the conclusions of this research. It

introduces the possibility of using this system as an inertial tracking system in the NPS Autonomous Underwater Vehicle (AUV) project [YUN97] and the body suit project [ZYDA97]. Recommendations are made for future work and ways to improve the quaternion attitude estimation filter.

II. BACKGROUND

A. INTRODUCTION

A basic requirement in virtual environments (VE) is the tracking of objects, especially humans. Tracking of humans creates a convenient human machine interface to the virtual environment. If the user is to interact in a natural way with a virtual environment, then the use of standard 2D devices becomes unacceptable [FREY96]. 3D user interfaces in a virtual environment require the use of devices that are special to that environment. There are many 3D user interface systems available; however, they all have unique problems in real-time applications. This chapter presents current motion tracking devices, their usage in virtual environments, and their effectiveness when applied to real-time human motion tracking.

B. REQUIREMENTS FOR SPATIAL TRACKERS

Tracking devices allow a virtual reality system to display the x, y and z position and the yaw, pitch and roll orientation of a tracked object or a human body part. The primary purpose of any tracking device is to provide an intuitive interface between human and machine [HENA97]. Human machine interfaces include all the devices used in a virtual environment system to present information to the human or to sense the actions or responses of the human [DURL95]. Before selecting an appropriate tracking device, it is necessary to determine the characteristics and behaviors of the tracked objects.

For normal arm movements during reaching, a fast motion is accomplished in about 0.5 seconds, wrist tangential velocities are about 3 m/s and the accelerations are about 5-6 g. For the fastest arm motion such as throwing a baseball, good pitchers release the ball at 37 m/s and accelerate their hands at more than 25 g. Motion

bandwidths of normal arm movements are around 2 Hz; the fastest hand motions are at around 5-6 Hz. The frequency content of normal arm motion can be defined as 5 Hz with a sampling rate of roughly 100 Hz [DURL95].

Head movements can be as fast as 1000 deg/s in yaw. Usual peak velocities are about 600 deg/s for yaw and 300 deg/s for pitch and roll. Tracker-to-host reporting rates must be at least 30 Hz. Delays of 60 ms or more between head motion and visual feedback are known to impair adaptation and the illusion of presence. Much smaller delays may cause simulator sickness [DURL95].

C. MOTION TRACKER PERFORMANCE

Several different tracking devices and technologies have been developed and applied to virtual environment applications. [MEYE92, SKOP96, HENA97] suggest some key measures by which tracking systems may be evaluated, namely;

- resolution and accuracy
- responsiveness
- robustness
- registration
- sociability

Resolution can be defined as the smallest change, which can be detected by a given tracking system [HENA97]. *Accuracy* is the sensor error range. The precision with which actions can be executed in the virtual world depends on the resolution and accuracy of a tracking device used. The *range* of a tracking device is the maximum distance between the sensor and source up to which the position and the orientation can

be measured with a specified error [BARA93]. Accuracy would also include sensor *drift*; i.e., the tendency of output to change without any change in input [SKOP96].

Responsiveness is a measure of the quickness with which new information is provided. It is determined by sampling rate, data rate, update rate, and latency [SKOP96]. *Sampling rate* is simply how often the sensor is checked for new data. *Data rate* is defined as the number of computed data points per second that the system can provide. Many systems will implement a much higher sampling rate than data rate in order to assure that new data is not missed. Filtering the sensor data takes time and will hinder real time updates. The rate at which the system can provide updated position and orientation data to the host computer is the *update rate.* Latency is perhaps the most important characteristic of responsiveness. The usefulness of tracking devices in virtual environments depends to a large degree on whether the computer can track the movements of the user fast enough to keep the virtual world synchronized with the user's actions. This ability is determined by the *lag* of the signal, and the sensor's update rate. The signal lag is the delay in time between a change of the position and orientation of the target being tracked and the report of the change to the computer. Lags above 50 ms are perceptible to the user and affect human performance. Typical update rates are between 30 and 60 updates per second [BARA93]. A system's lag is sometimes referred to as its *latency* and is one of the most important specifications of a tracking system [HENA97].

Robustness is a measure of the tracker's effectiveness in the presence of noise or other signal interference in the operating environment [SKOP96]. Types of interference include physical, metallic, electronic and acoustic [HENA97]. Depending on the technology used, sensors may be sensitive to metal objects, radiation from display

monitors, and various noise sources or objects coming between source and sensor. This interference can limit the effectiveness of tracking devices [BARA93].

Registration is the correspondence between a unit's actual position and orientation and its reported position and orientation over the domain of the working volume [SKOP96].

Finally, *sociability* describes a system's maximum range of operation, its working volume, and the ability to track multiple targets within that operating range. The working volume is that volume in which the tracker can accurately report position and/or orientation information [MEYE92, HENA97]. In addition to considering these performance factors, one might consider *availability, cost* and *ease of use* before actually selecting a tracking system [SKOP96].

D. TYPES OF MOTION TRACKERS

Most currently used tracking devices are active; that is, a generated source is attached to the object to be tracked or sensed by devices on the tracked object. In passive tracking, the target is monitored from a distance by one or several cameras [BARA93]. Current tracking devices are based on electromagnetic, acoustic, mechanical, optical and InfraRed (IR) technologies.

1. Mechanical Tracking Devices

Mechanical trackers measure changes in position and orientation by using jointed linkages directly connected to a point of reference. For body motion tracking, the point of reference can either be another part of the human body or a fixed surface near the human [SKOP96]. These trackers can be separated into two basic types, body based (*ex-oskeletal, goniometer* systems) and ground-based systems [DURL95]. Body based

systems are used to track the user's joint angles or end–effector positions relative to some other part of the body. Ground-based systems are attached to some surface near the user. Generally, the user grasps an implement whose position and orientation are tracked [SKOP96].

Since no external source is required, these sensors are not susceptible to external interference [HENA97], and are much less sensitive to their immediate environment than, say, electromagnetic trackers [HAND93]. The lag for mechanical trackers is very short, less than 5 ms, their update rate is fairly high, 300 updates per second, and they are very accurate [BARA93]. The physical linkages are well suited for providing haptic responses. Haptic responses are force feedback cues that enable the user to experience simulated exertion forces during a virtual environment simulation. These cues further enhance the realism of the environment and the immersion of the user [HENA97].

The fact that mechanical trackers are a system of physical linkages attached to the body or constantly held makes them cumbersome. The main disadvantage of mechanical trackers is that the user's motion is constrained by mechanical devices. These devices have a restrictive working volume and are usually not portable. They require a designated area for their use [HENA97].

Since they have moving parts, mechanical trackers can wear out after a period of time [HAND93]. Mechanical trackers tend to be accurate, responsive, robust and inexpensive, but they have poor sociability [MEYE92, SKOP96] and can be difficult and time consuming to calibrate [DURL95, PRAT94, SKOP96]. Applications requiring a limited range of motion and where user immobility is not a problem are best suited for this type of tracking system [MEYE92, HENA97].

2. Electromagnetic Tracking Devices

Electromagnetic trackers utilize artificially generated signals from the electromagnetic spectrum. In this case, the electromagnetic spectrum is defined in a narrow sense to mean radio and microwave frequencies [SKOP96].

An electromagnetic tracker comprises a transmitter and a receiver. A fluctuating magnetic field generated in the three orthogonal coils of a transmitter is picked up by three corresponding coils in the receiver. The variations in the received signal can be used to calculate the relative position and orientation of the receiver and transmitter. The fluctuating magnetic field may be Alternating Current or Direct Current [HAND93].

An alternative method recently proposed involves use of spread-spectrum ranging techniques [BIBL95, SKOP96]. This technique uses the measured time of flight of electromagnetic pulses to a receiver to determine range from a set of fixed transmitters. The concept is similar to that of the Global Positioning System (GPS). A minimum of three transmitters would be required to determine position via triangulation. A fourth transmitter would be required to ensure time can be accurately computed by the receiver. Transmitted signals would all occupy the same wide bandwidth and could utilize code division multiple access (CDMA) to preclude mutual interference [SKOP96].

Electromagnetic trackers have been commercially available for some time and are relatively inexpensive and easy to use [SKOP96]. These devices are generally very flexible due to the small size of the receiver (smaller than a 1” cube). Although the working volume is generally not very large, a few feet, it is usually possible to arrange various combinations of time-multiplexed transmitters and receivers to cover more space and track more objects [HAND93]. These systems tend to have good accuracy in a small

working space, with accuracy trailing off as distances from the transmitter increase [SKOP96].

Electromagnetic trackers suffer from several sources of error. Electromagnetic interference (EMI) from devices such as radios or display units can cause erroneous readings. Large objects made of ferrous metals can interfere with the electromagnetic field, again causing inaccuracies [HAND93].

Robustness is adversely effected by sensitivity to ferromagnetic objects in the vicinity, with alternating current based trackers being more suspectible than direct current based trackers. Alternating current systems tend to generate eddy currents in metallic objects, which then cause their own electromagnetic interference [SKOP96].

Adding power to the transmitter to increase the working volume can increase noise. Both AC and DC systems are adversely impacted by noise from power sources. Responsiveness is poor compared to other methods [SKOP96].

Latency is a serious problem with electromagnetic trackers. Up until recently the typical latency for an electromagnetic device was around 100 ms. This is mostly due to the filtering being performed on the data to remove noise. Newer versions of these devices have attempted to address this problem [HAND93]. Sociability is best in an environment without ferromagnetic occlusions, but is limited due to a small range of operation. Still, these systems can be very effective at shorter ranges [SKOP96]. Electromagnetic tracking works best in applications which require a limited working environment that is free of electromagnetic interference.

3. Acoustic Tracking Devices

Acoustic devices are sometimes called "ultrasonic trackers". Acoustic tracking is a fairly simple and well-understood technique [HAND93]. Such devices use high frequency ultrasonic sound waves for measuring the position and orientation of the target object by either phase-coherence tracking or time-of-flight (TOF) tracking [BARA93].

Phase coherence tracking works by measuring the difference in phase between sound waves emitted by a transmitter on the target and those emitted by a transmitter at a known reference point. Time-of-flight works by measuring the amount of time that it takes for sound emitted by transmitters on the target to reach sensors located at fixed positions in the environment. The transmitters emit sound at known times and only one transmitter is active at a time [BARA93]. One transmitter and three sensors are required for 3 DOF while three transmitters and three sensors are necessary for full 6 DOF tracking [HAND93].

Simple acoustic tracking devices can be constructed at low cost. They offer better range of operation than magnetic systems but can suffer severe effects from shadowing that can occur when tracked body parts are blocked by other objects [SKOP96]. Objects that move farther than half a wavelength in one update period will induce tracking error. The speed of sound in air varies with air temperature, pressure and humidity. Hence, calculations of distance may be incorrect due to environmental conditions unless steps are taken to account for these [HAND93]. Time-of-flight trackers typically suffer from a low update rate, brought about by the low speed of sound in air [BARA93]. Another common problem is that echoes of the sound signal will be reflected from acoustically "hard" surfaces, such as office walls, causing reception of "ghost" pulses at the sensor

and interference with other transmitted pulses [HAND93]. Time-of-flight tracking devices are vulnerable to spurious noise sources at any range [HENA97].

Phase-coherent systems enjoy many benefits over time-of-flight systems due to much higher data rates [MEYE92]. If the range is small, both systems offer good accuracy, responsiveness, and robustness. As range increases, data rates for time-of-flight systems decreases, causing responsiveness and robustness to decrease. Sociability of phase-coherent systems is better than that of time-of-flight systems due to larger working volumes [MEYE92].

4. Optical Tracking Devices

There are a variety of approaches to optical sensing for position tracking and mapping. Distance may be measured by triangulation, by time-of-flight or by interferometry. The passive light of the environment may be employed, structured light may be projected, light may be pulsed, or active or passive markers may be placed on a moving body. The different types of optical trackers can be broken into five categories; passive stereo vision systems, marker systems, structured light systems, laser radar systems, and laser interferometric systems [DURL95].

Passive stereo vision systems employ ambient light and square-array charge-coupled device (CCD) cameras [DURL95]. Multiple images from cameras with varying viewpoints are compared. Triangulation is then used to determine position [SKOP96]. An essential issue is to solve the correspondence problem of relating the same points in two different images. Passive stereo vision systems are unlikely to be useful in virtual environment in the near term, as robustness and accuracy are not yet comparable to active ranging systems [DURL95].

The stereo correspondence problem is solved in marker systems because a few, easily identifiable fiducial points are tracked on a moving body. To create very bright spots on the image a number of infrared light emitting diodes (IRED) are used. For detection, $1cm^2$ position sensing detectors (PSD), also called lateral effect photodiodes, are used. Multiple markers can be tracked to yield orientation and to follow multiple bodies simultaneously. Workspace volume is about $1m^3$. A fundamental problem with the use of PSD is reflection of IRED light from environmental surfaces [DURL95].

Structured light systems use a ray, plane of light, or a laser, to sweep across a scene. At each position of the plane, a light stripe is created, which is sensed by a two dimensional camera. The intersection of the known plane and the line of sight from the camera determines the three-dimensional coordinates. Another common method uses laser spot scanning of the scene. In this method either all or only portions of the scene may be scanned for data [DURL95].

Two of the most prevalent techniques that use lasers include laser radar and laser interferometry techniques. Laser radar works in the same way as acoustic ranging techniques, except that much higher data rates are possible. Techniques include both time-of-flight and phase shift. By scanning the entire scene, a three dimensional picture of the scene can be generated. Laser radar techniques are more appropriate for longer distances than laser techniques that use triangulation [DURL95].

Laser interferometry, uses a steered laser beam to track a reflector on the object being tracked. Phase-shift ranging and angular information from the steered system are used to determine position. An alternate method uses several lasers to track the reflector from different fixed positions to obtain range information. In this case, the intersection of

spheres whose radii are determined from the range information determines the location of the point being tracked. The problem with these techniques is that they provide only incremental displacement data and loss of signal via shadowing can be cause for re-calibration [DURL95].

5. Inertial Tracking Devices

Inertial tracking systems use a combination of linear acceleration, angular rate and magnetic sensors to determine rigid body orientation. Angular orientation is determined by integrating the output from the angular rate sensors. Angular rate sensors operate by using the differential combination of the outputs of two vibrating linear accelerometers. Angular rate sensor output has an error called drift. *Drift* is defined as the tendency of bias errors, inherent to the sensor, to cause increasing orientation estimation errors over time. This fundamental limitation makes angular rate sensors only a relatively short-term solution to determining a rigid body's spatial orientation [HENA97].

In order to compensate for the long-term errors introduced by the use of angular rate sensors, inertial systems utilize linear acceleration sensors called accelerometers. Accelerometers measure the gravity vector, relative to the object being tracked as well as the forced linear accelerations of the attached rigid body.

The third component of inertial systems is the magnetometer. The magnetometer is sensitive to the Earth's magnetic field and can sense rotations about the local vertical axis. Magnetometers must be used to correct drift errors in the horizontal plane [HENA97].

Originally these devices were used in guidance systems for airplanes and missiles and as such were cumbersome [HAND93]. Several companies have begun

manufacturing small devices, but a complete system is not commercially available yet for body tracking. These systems hold much promise for future application to the body tracking problem. Larger systems have shown that accuracy, resolution, response, robustness and registration requirements for human body tracking can be met by this technology. Although the technology is currently expensive, it is expected that costs will come down as devices are marketed. The greatest benefit of this type of system is its sociability. Whereas electromagnetic, acoustic and optic devices all require emissions from a source to track objects, inertial tracking systems are *sourceless*. This precludes the inevitable disastrous effects of occlusions and noise [SKOP96]. Pure inertial systems are capable of tracking only orientation. A hybrid system providing 3D location of one reference point on the body would be required to fully solve the human body tracking problem.

E. RELATED WORK

Various methods exist for tracking the rotations of an object and representing these rotations in a virtual environment. Orientation filters based on Euler Angles are not capable of tracking orientations through the vertical, due to the gimbal lock singularities and divide by zero errors in equations involving trigonometric functions [BACH97B].

A specific example of an orientation filter based on Euler Angles is the navigation filter used in the NPS AUV project [MCGH95, BACH96]. The filter incorporates inputs from an onboard Inertial Measuring Unit (IMU), a compass, and a water-speed sensor. Intermittent GPS fixes periodically provide accurate real-time navigational data [BACH96, HENA97]. A schematic representation of the attitude estimation part of the SANS navigation filter is shown in Figure 1.

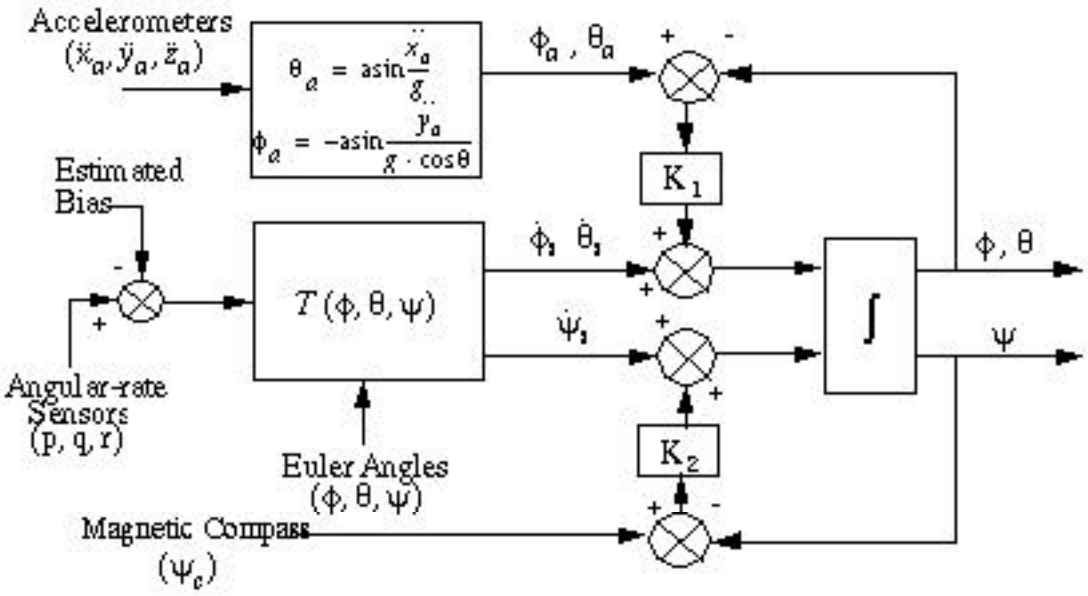

Figure 1: Euler Angle Attitude Filter [HENA97]

The SANS filter experiences singularities in two separate places. The accelerometer estimate of the roll angle and the angular rates of roll and azimuth become undefined at elevation angles of ±90 degrees.

In past research, [SKOP96] Euler Angles were used to track and to represent the human upper body. SKOP96 was forced to employ error-checking programming techniques in the software to avoid the singularities of an Euler Angle based orientation filter. He concluded that current electromagnetic trackers lack sufficient accuracy and registration to enable their use as a true six degree of freedom (DOF) tracker in human body tracking applications and called for the investigation of new tracking technologies to support the insertion of dismounted infantry into a virtual environment [ZYDA97].

In order to avoid singularities in human body tracking or in a navigation system, an alternative representation is needed for orientation filters. The quaternion attitude estimation filter was proposed by [MCGH96] as an alternative representation and improvement to the filters based on Euler Angles.

HENA97 developed the software necessary to support simulation of a quaternion filter using inertial sensors. His software was written in ANSI Common Lisp. The filter was tested with a computer simulated inertial tracker and used only static sensor readings. This research showed that the theory behind the quaternion attitude estimation filter was sound. Reduced computational complexity was achieved since the quaternion filter uses no trigonometric functions.

F. SUMMARY

Excepting inertial and spread-spectrum systems, the systems described in this chapter are currently available [SKOP96]. Mechanical techniques for tracking the upper body have been implemented and have been shown to be cumbersome and difficult to use. Acoustic trackers can provide potentially excellent accuracy and resolution together with a greater range of operation than magnetic trackers. Hybrid spread-spectrum ranging and inertial tracking systems have potential for providing increases in accuracy, response and range of operation over systems available now [SKOP96].

This chapter presented a brief overview of current tracking technologies and past research into body tracking systems and orientation filters. More detailed information about tracking systems can be found in [DURL95]. The next chapter presents representations of rigid body motion with Euler angles and quaternions, and a comparison between Euler angles and quaternions.

III. RIGID BODY ROTATION

A. INTRODUCTION

Computer animation and robotics both involve manipulating, rotating and translating moving objects in a 3D environment. Several different systems are used to describe positions and motions in space. These include Euler angles and quaternions. Each of these has particular advantages and disadvantages. This chapter will provide the basic information about Euler Angles and Quaternions.

B. EULER METHOD

The most popular method for describing orientation in Earth coordinates is the Euler Method. Using a sequence of three angles, the Euler Method provides an intuitive description of attitude. Although eleven other possibilities exist, these angles typically consist of the familiar azimuth angle, ψ, the elevation angle, θ, and the roll angle, ϕ, [CRAI89]. Euler angles specify three successive rotations to bring the Earth coordinates into alignment with the body coordinates [COOK92].

1. Rigid Body Rotation with Euler Angles

Euler's theorem states that any numbers of rotations of a rigid body about an Earth-fixed axis are equivalent to a single rotation about a single Earth-fixed axis. If all rotations are about the north(x), east(y) and down(z) axes, as depicted in Figure 2, the angles are called Euler Angles.

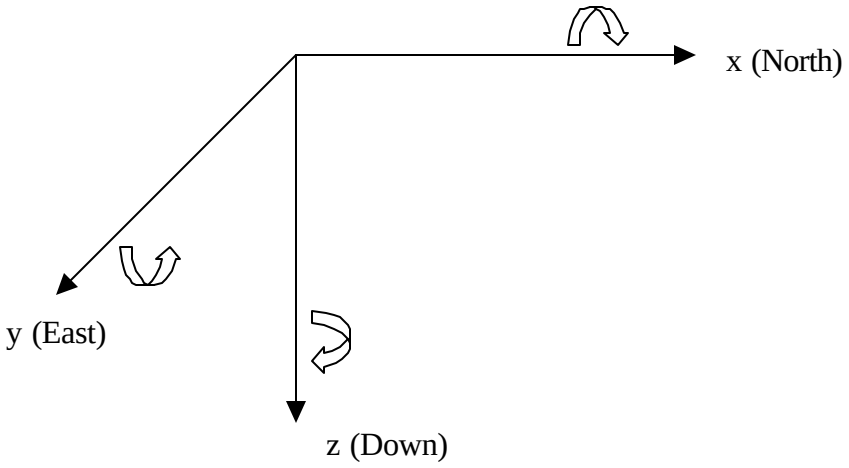

Figure 2: Right Handed Earth Coordinate Axis

As stated previously, there are twelve sets of rotations. The standard "azimuth, elevation, and roll" set is represented by the three rotation matrices given in Eq. 3.1. R_x represents a rotation about the x axis [CRAI89]. Similarly for R_z and R_y.

$$R = R_z R_y R_x \qquad \text{(Eq. 3.1)}$$

These names above and the symbols ψ, θ, and φ are reserved for the Euler angle set. The sign of a rotation is determined using the right hand rule. The ranges for the rotation angles are

$$y = \pm p \qquad q = \pm \frac{p}{2} \qquad j = \pm p \qquad \text{(Eq. 3.2)}$$

The x-axis rotation matrix (roll) is given by [CRAI89]

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \sin\varphi & \cos\varphi \end{bmatrix} \qquad \text{(Eq. 3.3)}$$

The y-axis rotation matrix (elevation, pitch) is given by

$$R_y(q) = \begin{bmatrix} \cos q & 0 & \sin q \\ 0 & 1 & 0 \\ -\sin q & 0 & \cos q \end{bmatrix} \qquad \text{(Eq. 3.4)}$$

The z-axis rotation matrix (azimuth, heading, yaw) is given by

$$R_z(y) = \begin{bmatrix} \cos y & -\sin y & 0 \\ \sin y & \cos y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{(Eq. 3.5)}$$

In general, every rotation matrix is a 3x3 matrix, but to make both translation and rotation calculations using matrices, similar 4x4 homogeneous matrices are used. A homogeneous rotation matrix in graphics, takes the form [FOLE97]

$$R = \begin{bmatrix} r & r & r & 0 \\ r & r & r & 0 \\ r & r & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(Eq. 3.6)}$$

Any rotation matrix can be put into this general form. Rotations of an object are made by multiplying these matrices with the homogenous coordinates of the object. In composing the results of successive rotations by matrix multiplication, the first rotation is associated with the right-most matrix. Depending on the order of the rotations, there exists more than one way to describe a given orientation.

Any rotation can be calculated by multiplying these individual matrices in the right order. For example a roll rotation, followed by an elevation and followed by a change in azimuth would be calculated by multiplying the matrices from right to left

$$R = R_z R_y R_x \quad \text{(Eq. 3.7)}$$

resulting in

$$R = \begin{bmatrix} \cos y \cos q & \sin j \sin q \cos y - \cos j \sin y & \sin j \sin y + \cos j \cos y \sin q \\ \sin y \cos q & \sin j \sin q \sin y + \cos j \cos y & \cos j \sin q \sin y - \cos y \sin j \\ -\sin q & \cos q \sin j & \cos j \cos q \end{bmatrix} \quad \text{(Eq.3.8)}$$

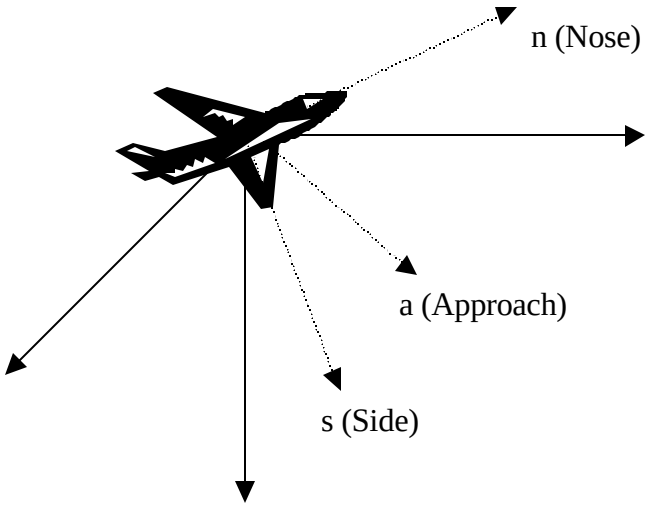

Figure 3: Body Coordinate Axes

2. Body Rates to Euler Rates

To rotate a body, its Rotational Velocity around the nose vector (n), the side vector (s) and the approach or belly vector (a) can be calculated, Figure 3. This rotational velocity can be presented in Earth coordinates as

$$E_{\omega} = \left[\omega_x \ \omega_y \ \omega_z\right]^T \qquad \text{(Eq. 3.9)}$$

and in body coordinates

$$B_{\omega} = \left[p \ q \ r\right]^T \qquad \text{(Eq. 3.10)}$$

where p is roll rate, q is pitch rate and r is yaw rate. These words and symbols are reserved for body rotations [MCGH93]. Euler rates are in Earth coordinates and p, q, and r are in body coordinates. Thus, the following must be noted.

- Roll rate, p ? roll Euler angle rate, $\dot{j}$
- Pitch rate, q ? elevation Euler angle rate, $\dot{q}$
- Yaw rate, r ? azimuth Euler angle rate, $\dot{y}$

The rotation rate in Earth coordinates is;

$$\mathrm{E}_{\omega} = \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} + \mathrm{R}_z \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \mathrm{R}_z \mathrm{R}_y \begin{bmatrix} \dot{\varphi} \\ 0 \\ 0 \end{bmatrix} \qquad \text{(Eq. 3.11)}$$

Evidently, the rotation in Body coordinates is;

$$B_w = \left[R_z R_y R_x \right]^{-1} E_w = \left[R_z R_y R_x \right]^{T} E_w \qquad \text{(Eq. 3.12)}$$

$$= R_x{}^T R_y{}^T R_z{}^T E_w = R_x{}^{-1} R_y{}^{-1} R_z{}^{-1} E_w \qquad \text{(Eq. 3.13)}$$

$$= R_x{}^T R_y{}^T R_z{}^T \begin{bmatrix} 0 \\ 0 \\ \dot{y} \end{bmatrix} + R_x{}^T R_y{}^T \begin{bmatrix} 0 \\ \dot{q} \\ 0 \end{bmatrix} + R_x{}^T \begin{bmatrix} \dot{j} \\ 0 \\ 0 \end{bmatrix} \qquad \text{(Eq. 3.14)}$$

Since

$$R_x{}^T R_y{}^T = \begin{bmatrix} \cos q & 0 & -\sin q \\ \sin j \sin q & \cos j & \sin j \cos q \\ \cos j \sin q & -\sin j & \cos j \cos q \end{bmatrix} \qquad \text{(Eq. 3.15)}$$

it follows that

$$R_x{}^T R_y{}^T \begin{bmatrix} 0 \\ \dot{q} \\ 0 \end{bmatrix} = \dot{q} \begin{bmatrix} 0 \\ \cos j \\ -\sin j \end{bmatrix} \qquad \text{(Eq. 3.16)}$$

Likewise,

$$R_x{}^T R_y{}^T R_z{}^T \begin{bmatrix} 0 \\ 0 \\ \dot{y} \end{bmatrix} = \dot{y} \begin{bmatrix} -\sin q \\ \sin j \cos q \\ \cos j \cos q \end{bmatrix} \qquad \text{(Eq. 3.17)}$$

and

$$R_x{}^T \begin{bmatrix} \dot{\varphi} \\ 0 \\ 0 \end{bmatrix} = \dot{\varphi} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \text{(Eq. 3.18)}$$

Thus

$$B_\omega = \dot{\psi}\begin{bmatrix} -\sin\theta \\ \sin\varphi\cos\theta \\ \cos\varphi\cos\theta \end{bmatrix} + \dot{\theta}\begin{bmatrix} 0 \\ \cos\varphi \\ -\sin\varphi \end{bmatrix} + \dot{\varphi}\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = (p\ q\ r)^T \qquad \text{(Eq. 3.19)}$$

and

$$B_\omega = \begin{bmatrix} -\dot{\psi}\sin\theta + \dot{\varphi} \\ \dot{\psi}\sin\varphi\cos\theta + \dot{\theta}\cos\varphi \\ \dot{\psi}\cos\varphi\cos\theta - \dot{\theta}\sin\varphi \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \qquad \text{(Eq. 3.20)}$$

In component form, this is

$$p = \dot{j} - \dot{y}\sin q \qquad \text{(Eq. 3.21)}$$

$$q = \dot{\psi}\sin\varphi\cos\theta + \dot{\theta}\cos\varphi \qquad \text{(Eq. 3.22)}$$

$$r = \dot{\psi}\cos\varphi\cos\theta - \dot{\theta}\sin\varphi \qquad \text{(Eq. 3.23)}$$

Solving these equations in terms of p, q and r, results in

$$\dot{q} = q\cos j - r\sin j \qquad \text{(Eq. 3.24)}$$

$$\dot{y} = r\sec q\cos j + q\sec q\sin j \qquad \text{(Eq. 3.25)}$$

$$\dot{j} = p + r\tan q\cos j + q\tan q\sin j \qquad \text{(Eq. 3.26)}$$

Equivalently, in matrix form;

$$\begin{bmatrix} \dot{j} \\ \dot{q} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} 1 & \tan q\sin j & \tan q\cos j \\ 0 & \cos j & -\sin j \\ 0 & \sec q\sin j & \sec q\cos j \end{bmatrix}\begin{bmatrix} p \\ q \\ r \end{bmatrix} \qquad \text{(Eq. 3.27)}$$

The equations from Eq. 3.21 through Eq. 3.27, also known as the gimbal rate equations and are quite commonly used in animation and simulation [COOK92]. The matrix in Eq. 3.27 is also referred to as the *T* matrix [FRAN69].

3. Problems with Euler Angles

The first problem results in divide-by-zero errors in equations involving trigonometric functions. Whenever $\cos\theta = 0$ $(\theta = \pm\pi/2)$

$$\tan\theta = \frac{\sin\theta}{\cos\theta} \quad , \quad \sec\theta = \frac{1}{\cos\theta} \qquad \text{(Eq. 3.28)}$$

are undefined.

A second problem exists when pitch, q, goes through the vertical;

$$q = \pm\frac{p}{2} \qquad \text{(Eq. 3.29)}$$

At that point, the roll and azimuth axes become coincidental. This is called gimbal lock singularity. "Gimbal lock" is a term derived from a mechanical problem that arises in the gimbal mechanism used to support a compass or gyroscope. The final rotation matrix depends on the order of multiplication. It is sometimes the case that the rotation in one axis will be mapped onto another rotation axis. Even worse, it may become impossible to rotate an object in a desired axis [WATT98].

Both these problems occur because Euler angles ignore the interaction of the rotations about separate axes. In truth, these rotations are not independent of each other [WATT98].

The only solution to going through a vertical orientation is to fake; i.e., fix the code so a division-by-zero error doesn't occur. These fixes usually prove less than satisfactory and make it impossible to track an object through all possible orientations [COOK92].

C. QUATERNION METHOD

An alternate method of describing orientation that has been gaining popularity in the graphics community is the use of unit quaternions. It is not a new method [COOK92]; quaternions have been around for more than 150 years.

The quaternion is a concept related to three dimensional vectors, but which allows the representation of operations such as rotations not directly representable with vectors [PERV82]. A quaternion is a four dimensional vector with an associated quaternion product. It is conventional to interpret a quaternion as a generalization of a complex number with a real number part and a vector part.

1. Quaternion Basics

The basis for quaternions are three imaginary "flags" i, j and k where [MCCA90]

$$i^2 = j^2 = k^2 = ijk = -1 \qquad \text{(Eq. 3.30)}$$

$$ij = -ji = k \qquad \text{(Eq. 3.31)}$$

$$jk = -kj = i \qquad \text{(Eq. 3.32)}$$

$$ki = -ik = j \qquad \text{(Eq. 3.33)}$$

There are several equivalent ways of writing quaternions in terms of their four components; one way is the Standard Quadrinomial Form [PERV82]:

$$Q = \{a + bi + gj + dk : a, b, g, d \text{ real}\} \qquad \text{(Eq. 3.34)}$$

There are three other commonly used quaternion notations.

Linear combination of four components:

$$q = w + xi + yi + zk \qquad \text{(Eq. 3.35)}$$

Four dimensional vector:

$$q = (w\ x\ y\ z) \qquad \text{(Eq. 3.36)}$$

Scalar with a vector "imaginary part":

$$q = (w, v) \qquad \text{(Eq. 3.37)}$$

where w is a scalar quantity and v is a vector. Quaternion addition is defined in the same manner as normal vector addition

$$q_1 + q_2 = ((w_1 + w_2)(x_1 + x_2)(y_1 + y_2)(z_1 + z_2)) \qquad \text{(Eq. 3.38)}$$

If s is a scalar, then scalar multiplication with a quaternion is defined as

$$sq = (sw, sv) \qquad \text{(Eq. 3.39)}$$

Quaternion multiplication can be defined using Eq. 3.30 through Eq. 3.33. Let $q_1 = w_1 + ix_1 + jy_1 + kz_1$ and $q_2 = w_2 + ix_2 + jy_2 + kz_2$. Then

$$\begin{aligned} q_1 * q_2 = {} & (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) + \\ & i(x_1 w_2 + w_1 x_2 - z_1 y_2 + y_1 z_2) + \\ & j(y_1 w_2 + z_1 x_2 + w_1 y_2 - x_1 z_2) + \\ & k(z_1 w_2 - y_1 x_2 + x_1 y_2 + w_1 z_2) \end{aligned} \qquad \text{(Eq. 3.40)}$$

Note that COOK92 has sign errors in quaternion multiplication. The same result can be accomplished through a vector dot product, vector cross product and vector scalar multiplications [MCGH98A].

$$q_1 * q_2 = (w_1 w_2 - v_1 \bullet v_2, w_1 v_2 + w_2 v_1 + v_1 \times v_2) \qquad \text{(Eq. 3.41)}$$

Because of the vector cross product, the quaternion product is not commutative. Some mathematical facts related to quaternions are as follows [SHOE94, CRAI89]

Quaternion addition rules;

$$q_1 + q_2 = q_2 + q_1 \qquad \text{(Eq. 3.42)}$$

$$(q_1 + q_2) + q_3 = q_1 + (q_2 + q_3) \qquad \text{(Eq. 3.43)}$$

$$q + 0 = 0 + q \quad \text{(Eq. 3.44)}$$

$$q + (-q) = (-q) + q = 0 \quad \text{(Eq. 3.45)}$$

Quaternion multiplication rules;

$$(q_1 q_2) q_3 = q_1 (q_2 q_3) \quad \text{(Eq. 3.46)}$$

$$1q = q1 = q \quad \text{(Eq. 3.47)}$$

$$qq^{-1} = q^{-1} q = 1 \quad \text{(Eq. 3.48)}$$

$$\text{if } q_1 q_2 = 0, \text{ then either } q_1 = 0 \text{ or } q_2 = 0 \quad \text{(Eq. 3.49)}$$

$$(q_1 + q_2) q_3 = q_1 q_3 + q_2 q_3 \quad \text{(Eq. 3.50)}$$

The quaternion conjugate;

$$q^* = (w, -v) = (w, \ -xi \ -yj \ -zk) = (w \ -x \ -y \ -z) \quad \text{(Eq. 3.51)}$$

The quaternion Norm;

$$\mathrm{N(q)} = \mathrm{qq^*} = \mathrm{w}^2 + |\mathrm{v}|^2 = \mathrm{w}^2 + \mathrm{v} \bullet \mathrm{v} = |\mathrm{q}|^2 \quad \text{(Eq. 3.52)}$$

The magnitude of a quaternion is the square root of its norm.

$$M(q) = \sqrt{N(q)} \quad \text{(Eq. 3.53)}$$

The quaternion inverse is evidently

$$q^{-1} = \frac{q^*}{N(q)} \quad \text{(Eq. 3.54)}$$

and for a unit quaternion

$$q^{-1} = q^* \quad \text{(Eq. 3.55)}$$

which is easier to compute than the inverse of a matrix.

The normalized unit quaternion is given by

$$q_{normalized} = \frac{q}{\sqrt{N(q)}} = \frac{q}{\sqrt{qq^*}} = \frac{q}{M(q)} \quad \text{(Eq. 3.56)}$$

Any scalar can be represented as a quaternion.

$$q = (s\ 0\ 0\ 0) = (s,0) \quad \text{(Eq. 3.57)}$$

Any three dimensional vector can be represented as a quaternion.

$$q = (0\ x\ y\ z) = (0, v) \quad \text{(Eq. 3.58)}$$

As an alternative, a quaternion can be represented as

$$q = w + \vec{v} \quad \text{(Eq. 3.59)}$$

The vector part of quaternion can be rewritten as

$$\vec{v} = s\vec{r} = \vec{r}s \quad \text{(Eq. 3.60)}$$

where,

$$s = |\vec{v}| = \sqrt{(x^2 + y^2 + z^2)} \quad \text{(Eq. 3.61)}$$

and r is the unit vector given by

$$\vec{r} = \frac{\vec{v}}{s} = \frac{\vec{v}}{|\vec{v}|} \quad \text{(Eq. 3.62)}$$

With this notation a quaternion becomes

$$q = w + \vec{r}s \quad \text{(Eq. 3.63)}$$

Another representation, the quaternion exponential form is defined as [MCGH98A];

$$e^q = e^{w+\vec{r}s} = e^w e^{\vec{r}s} \quad \text{(Eq. 3.64)}$$

$$e^{\vec{r}s} = \cos s + \vec{r}\sin s \quad \text{(Eq. 3.65)}$$

The conjugate becomes

$$q^* = w - \vec{r}s \qquad \text{(Eq. 3.66)}$$

2. Rigid Body Rotation with Quaternions

The orientation of a rigid body can be described as a rotation (q) about a single inclined axis (u). Constraining the axis (u) to be of unit magnitude, the unit quaternion (q) representing this rotation is [MCGH97]

$$q = \left(Cos\frac{q}{2}, uSin\frac{q}{2} \right) \qquad \text{(Eq. 3.67)}$$

where u is a unit vector describing the axis about which the vector p is to be rotated through an angle q [BACH97B]. The rotation of a vector, p by a quaternion, q is defined as [SHOE85, FUND96]

$$P_{rotated} = qpq^{-1} \qquad \text{(Eq. 3.68)}$$

The quaternion rotation defined in Eq. 3.68, rotates the vector's perpendicular component twice the angle which is perpendicular to the rotation axis, and leaves the vector's parallel component unchanged. To obtain the desired rotation half of the rotation angle is used in the construction of the unit quaternion given by Eq. 3.67 [MCGH98A].

Every rotation has two representations in quaternion space, namely q and $-q$, with the same effect [WATT98]. By using this topological oddity, Eq. 3.68 can be rewritten as

$$P_{rotated} = qpq^{-1} = (-q)\,p(-q)^{-1} \qquad \text{(Eq. 3.69)}$$

The product of two unit quaternions is always of unit magnitude. The product q_2q_1 produces a single quaternion describing an orientation achieved by applying q_2 relative to the orientation described by q_1, where both quaternions are expressed in Earth

coordinates. If rotations are described by quaternions in body coordinates, then the order of the product is reversed.

It should be noted that, unlike Euler angles, quaternion rotations require only two trigonometric functions to rotate a vector and experience no singularities at any angle of rotation.

To illustrate quaternion rotation through an example, let p=(0 1 0 0) describe a body oriented along the x-axis, wings level, headed to north. A positive 90° rotation about the y-axis can be represented by

$$q = (Cos45°, vSin45°) \quad \text{(Eq. 3.70)}$$

$$= (0.707\ 0\ 0.707\ 0) \quad \text{(Eq. 3.71)}$$

where v = (0 1 0).

The following product rotates the body with an orientation described by p 90° about the y-axis

$$P_{rotated} = qpq^{-1} \quad \text{(Eq. 3.72)}$$

$$= (0\ 0\ 0\ -1) \quad \text{(Eq. 3.73)}$$

which represents an orientation along the –z-axis, in which the body is pointing straight up [BACH97B].

3. Derivation of Quaternion Rates from Body Rates

Angular rates p, q and r, can be used to find the derivative of the orientation, $\dot{q}$, relative to an Earth fixed coordinate system.

For small ?

$$cos\frac{\theta}{2} \cong 1 \qquad sin\frac{\theta}{2} \cong \frac{\theta}{2} \quad \text{(Eq. 3.74)}$$

Thus

$$q = \left(\cos\frac{q}{2}, v\sin\frac{q}{2} \right) \cong \left(1, v\frac{q}{2} \right) \quad \text{(Eq. 3.75)}$$

Assuming ? changes linearly with time, the orientation expressed by q as a function of time becomes, for small t:

$$q(t) = \left(1, \frac{1}{2} v\dot{q}t \right) \quad \text{(Eq. 3.76)}$$

The vector $v\dot{q}$, expresses an angular rate of $\dot{q}$ about a vector v in body coordinates. Thus

$$v\dot{\theta} = \begin{pmatrix} p & q & r \end{pmatrix} \quad \text{(Eq. 3.77)}$$

and *q(t)* becomes

$$q(t) = \left(1, \frac{1}{2} pt \ \frac{1}{2} qt \ \frac{1}{2} rt \right) \quad \text{(Eq. 3.78)}$$

Taking the derivative of *q(t)* with respect to time produces

$$\frac{d}{dt} q(t) = \dot{q} = (0, \frac{1}{2} p \ \frac{1}{2} q \ \frac{1}{2} r) \quad \text{(Eq. 3.79)}$$

$$= \frac{1}{2}(0 \ p \ q \ r) \quad \text{(Eq. 3.80)}$$

$$= \frac{1}{2} \mathrm{B}_w \quad \text{(Eq. 3.81)}$$

If q_1 is the initial orientation in Earth coordinates and q_2 is a second rotation in body coordinates then q_3 is the composite of the two rotations

$$q_3 = q_1 q_2 \quad \text{(Eq. 3.82)}$$

By the product rule

$$\dot{q}_3 = \dot{q}_1 q_2 + q_1 \dot{q}_2 = q_1 \dot{q}_2 = \frac{1}{2} q_1 B_w \qquad \text{(Eq. 3.83)}$$

In general [BACH97B]

$$\dot{q} = \frac{1}{2} q B_\omega = \frac{1}{2} q(0\ p\ q\ r) \qquad \text{(Eq. 3.84)}$$

This general formula can be used with Euler integration to achieve a smooth rotation. Eq. 3.84, also avoids the "branch cut" problem of the Euler angles. After every full 360 degree rotation, the quaternion representing the rotation will switch to its negative [MCGH98A] as presented in Eq. 3.69.

4. Concatenating Rotations

Suppose q_1 and q_2 are unit quaternions representing two rotations relative to Earth coordinates, and it is desired to perform q_1 first and then q_2. To do this, we apply q_2 to the result of q_1, regroup the product using associativity, and find that the composite rotation is represented by the quaternion $q_2 * q_1$.

$$q_2 * (q_1 * P * q_1^{-1}) * q_2^{-1} = (q_2 * q_1) * P * (q_1^{-1} * q_2^{-1}) \qquad \text{(Eq. 3.85)}$$

$$= (q_2 * q_1) * P * (q_2 * q_1)^{-1} \qquad \text{(Eq. 3.86)}$$

5. Computing Rotation Matrices from Quaternions

The only time we need to compute a matrix is when we want to transform the object using a homogenous transformation. Alternatively, rotation and translation can be handled separately, eliminating the need for computing the rotation matrix [SKOP96]. Matrix multiplication requires many more operations than a quaternion product. Thus, efficiency and numerical accuracy is improved through the use of quaternions rather than

matrices. A rotation matrix for the given rotation can be calculated from a quaternion representing that same rotation. The general rotation matrix is as follows;

$$Q_{matrix} = \begin{bmatrix} w^2 + x^2 - y^2 - z^2 & 2xy + 2wz & 2xz - 2wy & 0 \\ 2xy - 2wz & w^2 - x^2 + y^2 - z^2 & 2xy + 2wx & 0 \\ 2xz + 2wy & 2yz - 2wx & w^2 - x^2 - y^2 + z^2 & 0 \\ 0 & 0 & 0 & w^2 + x^2 + y^2 + z^2 \end{bmatrix} \quad \text{(Eq. 3.87)}$$

The simplified version of this matrix for unit quaternions is [SHOE85];

$$Q_{matrix} = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy + 2wz & 2xz - 2wy & 0 \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2xy + 2wx & 0 \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(Eq. 3.88)}$$

Using this matrix, the rotations of a vector P can be computed as [COOK92]

$$P_{rotated} = P * Q_{matrix} \quad \text{(Eq. 3.89)}$$

6. Slerp

The use of linear interpolation between two unit quaternions produces non-unit quaternions without normalization. Spherical linear interpolation (*slerp*) between two unit quaternions is a natural generalization of linear interpolation [SHOE85] and is obtained by

$$slerp(q_1, q_2, u) = q_1 \frac{\sin(1-u)\Omega}{\sin\Omega} + q_2 \frac{\sin u\Omega}{\sin\Omega} \quad \text{(Eq. 3.90)}$$

$$q_1 \bullet q_2 = \cos\Omega \quad \text{(Eq. 3.91)}$$

where u is between 0 and 1, and O is the interpolation angle between two key quaternions. For the opposite direction, the interpolation angle becomes 2p - O [WATT98].

Given any two key quaternions, p and q, there exists two possible arcs along which we can move. One of these arcs is shorter. A method for finding the shortest interpolation between the quaternion pairs p and q, and p and $-q$ is as follows. First find the magnitude of their difference, that is $(p-q)\bullet(p-q)$, and then compare this to the magnitude of the difference when the second quaternion is negated, that is $(p+q)\bullet(p+q)$. If the former is smaller, then we are already moving along the smaller arc and nothing needs to be done. If, however the second is smaller, then we replace q by $-q$ and proceed [WATT98].

D. COMPARISON OF EULER ANGLES AND QUATERNIONS

Quaternions and Euler angles each have their own advantages and disadvantages. The most significant advantage of quaternions is that no singularity exists when the elevation angle (?) passes through ±p/2. Any orientation or rotation can be represented by quaternions. In the Euler Method, roll and azimuth Euler angle rates become undefined due to division by zero. Truncating the angles at ±p/2 will avoid this problem. However, this truncating will result in some skipping during the rotation [COOK92].

Quaternion rotation avoids the “branch cut” problem of Euler angles. When using quaternions to rotate an object, after every full rotation the quaternion representing the rotation will switch to its negative.

Quaternions can be computed directly from the dynamic equations, bypassing the computation of transcendental functions necessary in computing Euler angles. This direct arithmetic operation reduces the cost of computation compared to matrix multiplication.

Quaternions are compact and simple. However, there are three difficulties with using quaternions. First, each orientation of an object can actually be represented by two quaternions. Since rotation about the axis, *v* by an angle ? is the same as rotation about –*v* by the angle –?; the corresponding quaternions are antipodal points on the sphere in 4D. Second, orientations and rotations are not exactly the same thing. In an animation, 360º rotation is very different from a rotation of 0º. With the first one, the model will be animated for a full rotation. In the second one, no animation will be executed. After the rotation however, the same quaternion (1 0 0 0) or (-1 0 0 0) represents both. Thus specifying multiple rotations with quaternions requires intermediate control points [FOLE97].

The third difficulty is that quaternions provide an isotropic method for rotation. The interpolation is independent of everything except the relation between the initial and final rotations. This is useful for interpolating orientations of tumbling bodies, but not for interpolating the orientations of a virtual camera in a scene. Humans strongly prefer cameras to be upright, and are profoundly disturbed by tilted cameras [FOLE97]. By its very nature, the notion of a preferred direction cannot easily be built into the quaternion representation [WATT98]. Quaternions have no such preferences, and therefore should not be used for camera interpolation. The lack of an adequate method for interpolating complex camera motion has led to many computer animations having static cameras or very limited camera motion [FOLE97].

E. SUMMARY

This chapter presented a brief overview of two methods, namely Euler and Quaternion methods, used to represent the rotations and orientations of a rigid body. This

chapter also provided some introductory information about quaternion operations. The next chapter introduces the theory and the mathematical formulation of a quaternion attitude estimation filter.

IV. QUATERNION ATTITUDE ESTIMATION FILTER

A. INTRODUCTION

The purpose of any filter is to separate one thing from another [BROW97]. The main purpose of the Quaternion Attitude Estimation Filter is to combine independent noisy inertial measurements to determine the orientation of a tracked object. The Quaternion Attitude Estimation Filter uses different types of instruments to get the measurement values and implements a nonlinear complimentary filter. Orientation estimates are corrected by minimizing the mean square measurement error. The Quaternion Attitude Estimation Filter is designed as a complimentary filter. However, with the addition of bias estimation, it can no longer be considered a complimentary filter. This chapter will present the mathematical theory of the quaternion filter, the filter linearization theory, bias estimation considerations and the white noise effects on the filter.

B. QUATERNION ATTITUDE ESTIMATION FILTER

The Quaternion Attitude Estimation Filter was proposed by [MCGH96] as an alternative representation and improvement to filters based on Euler Angles. The Quaternion Attitude Estimation Filter is designed to track human limb segments through all orientations as part of an inertial tracking system. It uses three different types of sensors to obtain the information about the orientation of a tracked object. These sensors are a three-axis accelerometer, a three-axis angular rate sensor and a three-axis magnetometer. These sensor inputs appear in Figure 4 as $(\ddot{x}\ \ddot{y}\ \ddot{z})$, $(p\ q\ r)$, and $(b_1\ b_2\ b_3)$, respectively.

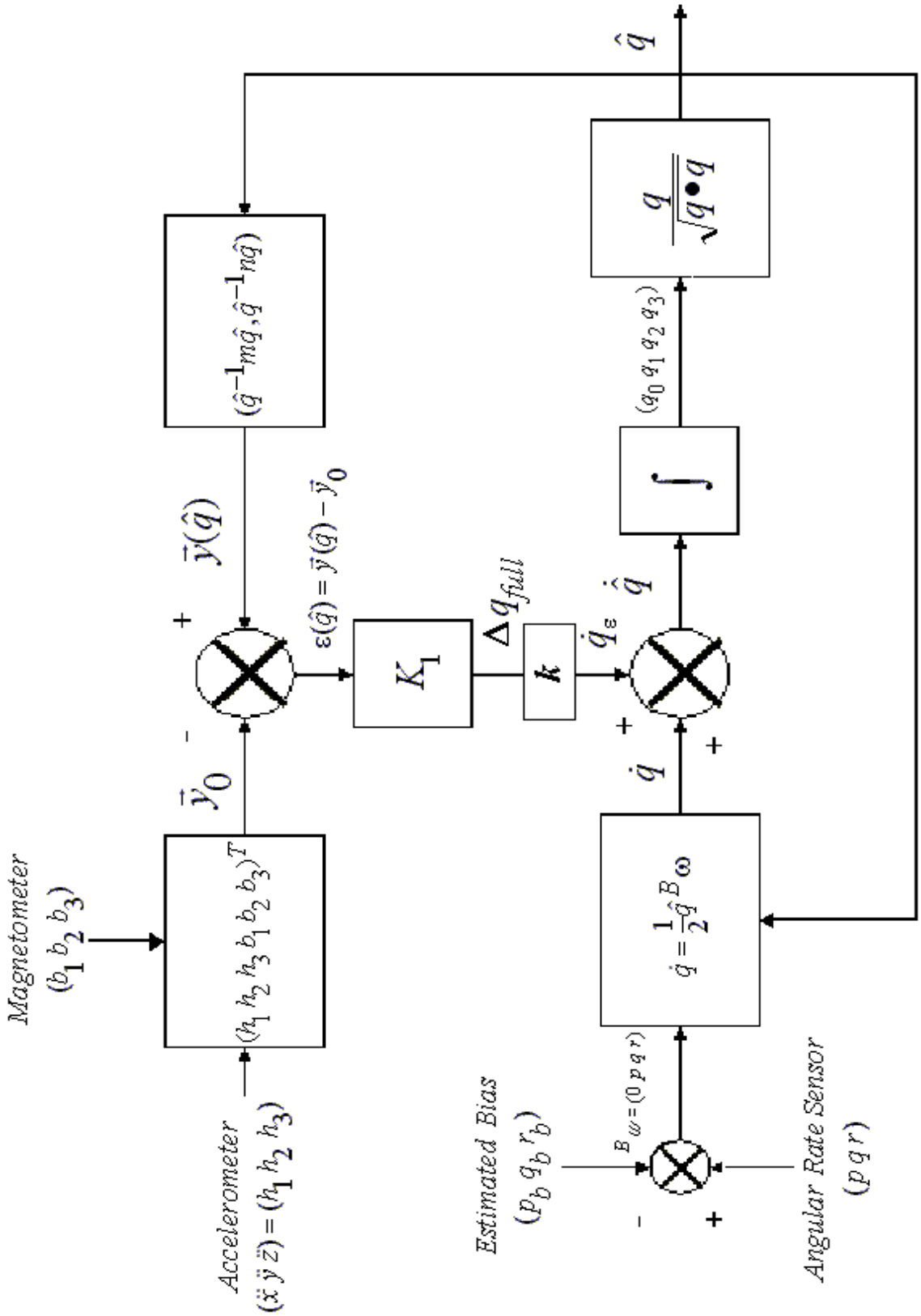

Figure 4: Quaternion Attitude Estimation Filter

The three-axis accelerometer measures the combination of forced linear accelerations and the reaction force due to gravity, $\vec{a}_{measured} = \vec{a} - \vec{g}$ [FOXL94].

The three-axis angular rate sensors measure angular velocity. The angular rate sensor experiences drift errors over time. To correct the measured angular rate sensor data in both the horizontal and vertical planes, accelerometer and magnetometer information is required.

Normally, the direction of the Earth's gravity vector, *m*, expressed as a unit vector, is down. For low frequency considerations, the linear accelerations of a body average to zero. That is, in the long term, the accelerometer senses only those accelerations due to gravity [BACH97B]. Thus, on the average, the three-axis accelerometer returns the gravity vector or the local vertical, $\vec{a}_{measured} = -\vec{g}$ [HENA97]. The down unit vector in quaternion form can be represented as

$$m = [0\ 0\ 0\ 1] \qquad \text{(Eq. 4.1)}$$

The three-axis magnetometer measures the Earth's magnetic field, *b*, in body coordinates [HENA97]. The main purpose of the magnetometer is to sense the drift error of the angular rate sensor about the vertical axis. This can not be sensed by the accelerometer [HENA97]. The Earth's unit magnetic field, *n*, at any location is known and can be calculated or looked up [BACH97B]. The Earth's unit magnetic field can be represented as a unit vector by;

$$n = [0\ n_1\ n_2\ n_3] \qquad \text{(Eq. 4.2)}$$

The characteristics of the Earth's magnetic field in the local area must be determined to find *n*, which incorporates the declination and the dip angle. Magnetic

declination is defined as the differrence between the north compass heading and the true geographic north at a given location on the Earth's surface. Monterey, CA requires a correction of 15°. The lines of the Earth's magnetic field are parallel to the surface at the equator. However, as one approaches the magnetic poles, they become increasingly vertical. Dip angle is the correction of the measure for the downward deflection of the local magnetic field. In Monterey, a correction of -60° is applied [HENA97].

C. DERIVATION OF THE FILTER

The Quaternion Attitude Estimation Filter takes normalized measurements from the three-axis accelerometer and from the three-axis magnetometer as shown in Figure 5 [HENA97].

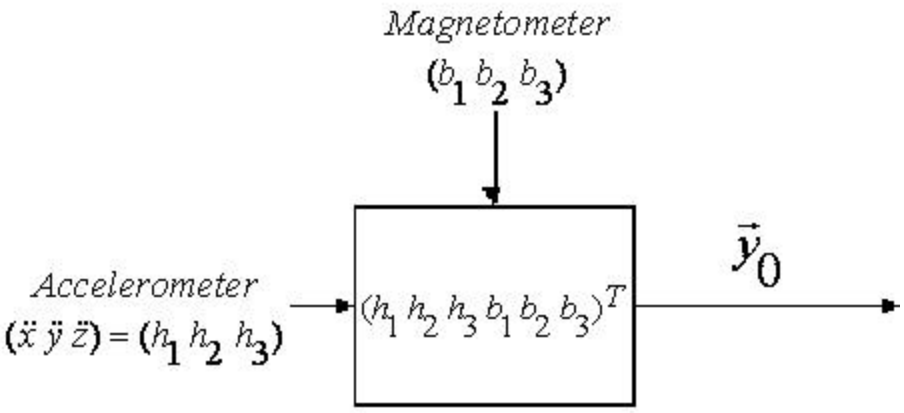

Figure 5: Upper Left Part of Quaternion Attitude Estimation Filter

The accelerometer returns the local vertical, *h* and the magnetometer returns the Earth's local magnetic field, *b*. This information is expressed in body coordinates as 3-dimensional unit vectors in the form of the pure imaginary unit quaternions given in Eq. 4.3 and Eq. 4.4.

$$h = [0\ h_1\ h_2\ h_3] \qquad \text{(Eq. 4.3)}$$

$$b = [0\, b_1\, b_2\, b_3] \qquad \text{(Eq. 4.4)}$$

The filter combines these vectors to get the measured vector, $\vec{y}_0$ given by;

$$\vec{y}_0 = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}_{6x1} \qquad \text{(Eq. 4.5)}$$

After forming the measured vector, the filter calculates the computed measurement vector, $\vec{y}(\hat{q})$, using the estimated orientation quaternion, $\hat{q}$. The Earth's unit gravity vector, *m*, and the local magnetic field vector, *n*, are rotated into body coordinates in order to permit comparison with the measured orientation vector. The estimated orientation of body, $\hat{q}$, determines $\hat{h}$ and $\hat{b}$ by;

$$\hat{h} = \hat{q}^{-1} m \hat{q} \qquad \text{(Eq. 4.6)}$$

and

$$\hat{b} = \hat{q}^{-1} n \hat{q} \qquad \text{(Eq. 4.7)}$$

Where the real or scalar part, *w*, is always zero. The estimated orientation quaternion is set when the system is initialized. The computed measurement vector is given by;

$$\vec{y}(\hat{q}) = (\hat{q}^{-1} m \hat{q}, \hat{q}^{-1} n \hat{q}) \qquad \text{(Eq. 4.8)}$$

By removing the w values of $\hat{h}$ and $\hat{b}$, the computed measurement vector can be constructed as;

$$\vec{y}(\hat{q}) = \begin{bmatrix} \hat{h}_1 \\ \hat{h}_2 \\ \hat{h}_3 \\ \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \end{bmatrix}_{6x1} \qquad \text{(Eq. 4.9)}$$

The error is computed by taking the difference between the measured vector and the computed vector and is given by

$$e(\hat{q})_{6x1} = \vec{y}(\hat{q})_{6x1} - \vec{y}0_{6x1} \qquad \text{(Eq. 4.10)}$$

or

$$e(\hat{q}) = \begin{bmatrix} \hat{h}_1 - h_1 \\ \hat{h}_2 - h_2 \\ \hat{h}_3 - h_3 \\ \hat{b}_1 - b_1 \\ \hat{b}_2 - b_2 \\ \hat{b}_3 - b_3 \end{bmatrix}_{6x1} \qquad \text{(Eq. 4.11)}$$

To correct this error an iterative method must be applied. In [HENA97], two different iteration methods were tested in simulation namely; Gradient Descent and Gauss-Newton. The Gradient descent method linearly converged, but needed dozens of steps to arrive at a satisfactory result. The Gauss-Newton method converged quadratically and needed less iteration. Based on this result, Gauss-Newton was used for this thesis.

For the Quaternion Attitude Estimation Filter the error correction vector which approximates Gauss-Newton's iteration formula [MCGH63] is given in Eq. 4.12;

$$\Delta\vec{q}_{4x1} = -\frac{1}{2}\left[X^T X\right]^{-1}\tilde{\mathrm{N}}\vec{f} \qquad \text{(Eq. 4.12)}$$

where the X matrix is given by

$$X_{ij}^T = \begin{bmatrix} \frac{\partial y_i}{\partial \hat{q}_j} \end{bmatrix}_{4x6} \qquad \text{(Eq. 4.13)}$$

Appendix A presents the derivation of the X matrix given in Eq. 4.13.

The least-squares criterion function is given by

$$f(\hat{q})_{1x1} = e(\hat{q})^T{}_{1x6}\, e(\hat{q})_{6x1} \qquad \text{(Eq. 4.14)}$$

where

$$e(\hat{q})^T = \left[(\hat{h}_1 - h_1)(\hat{h}_2 - h_2)(\hat{h}_3 - h_3)(\hat{b}_1 - b_1)(\hat{b}_2 - b_2)(\hat{b}_3 - b_3)\right]_{1x6} \qquad \text{(Eq. 4.15)}$$

Substituting Eq. 4.15 into Eq. 4.14 yields

$$f(\hat{q})_{1x1} = \left[(\hat{h}_1 - h_1)^2 + (\hat{h}_2 - h_2)^2 + (\hat{h}_3 - h_3)^2 + (\hat{b}_1 - b_1)^2 + (\hat{b}_2 - b_2)^2 + (\hat{b}_3 - b_3)^2\right] \qquad \text{(Eq. 4.16)}$$

The gradient of the error criterion function is defined as;

$$\nabla f(\hat{q}) = \left(\frac{\partial f}{\partial \hat{q}_0}\frac{\partial f}{\partial \hat{q}_1}\frac{\partial f}{\partial \hat{q}_2}\frac{\partial f}{\partial \hat{q}_3}\right)^T = 2X^T{}_{4x6}\, e(\hat{q})_{6x1} \qquad \text{(Eq. 4.17)}$$

The derivation of this gradient is given in Appendix B.

Substituting Eq. 4.17 into Eq. 4.12 yields

$$\Delta\vec{q}_{4x1} = -\frac{1}{2}\left[X^T X\right]^{-1} 2X^T e(\hat{q}) \qquad \text{(Eq. 4.18)}$$

$$= -\left[X^T X\right]^{-1}_{4x4} X^T_{4x6}\varepsilon(\hat{q})_{6x1} \qquad \text{(Eq. 4.19)}$$

The $\dot{q}_e$ in Figure 4 represents the output from the top-half of the filter, and it will be used to correct the computed rate quaternion derivation, $\dot{q}$. To find the output of the top-half of the filter the error correction vector, $\Delta\vec{q}$, must be multiplied by a constant gain, *k*

$$\dot{q}_\varepsilon = k\Delta\vec{q} \qquad \text{(Eq. 4.20)}$$

A scalar multiplier, a, is used when dealing with data corrupted by noise

$$a = k\Delta t \qquad \text{(Eq. 4.21)}$$

1. Linearization

Δq_{full} is the correction to $\hat{q}$ based upon the Gauss-Newton iteration formula. Assuming there is no measurement noise, and that the computation of Δq_{full} is exact, it follows that [MCGH98C]

$$\Delta\vec{q}_{4x1} = q_{true} - \hat{q} \qquad \text{(Eq. 4.22)}$$

$$\Delta q_{full} = q_{true} - \hat{q} \qquad \text{(Eq. 4.23)}$$

The signal flow diagram (SFG) representing this linearized system is presented in Figure 6.

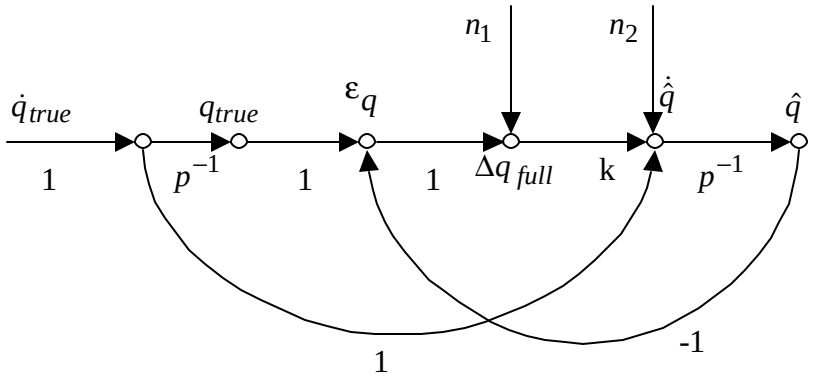

Figure 6: Linearized System Isolating Error

Applying Mason's formula [KUO95] to Figure 6 to obtain the transfer function produces

$$\frac{\hat{q}}{\dot{q}_{true}} = \frac{kp^{-2} + p^{-1}}{1 + kp^{-1}} = \frac{p^{-1}(1 + kp^{-1})}{1 + kp^{-1}} = p^{-1} \qquad \text{(Eq. 4.24)}$$

thus

$$\hat{q} = p^{-1}\dot{q}_{true} = q_{true} \qquad \text{(Eq. 4.25)}$$

This result shows that regardless of the value of k, if n_1 and n_2 are zero, then $\hat{q} = q_{true}$. This is as expected for a complimentary filter.

2. Response to Initial Condition Error

Suppose the sensors are static and $\hat{q}$ is not correctly initialized then

$$q_{true} = (1\ 0\ 0\ 0) \qquad \text{(Eq. 4.26)}$$

$$\hat{q}_0 = (1\ \partial_x\ \partial_y\ \partial_z) \qquad \text{(Eq. 4.27)}$$

If the noise sources n_1 and n_2 are both zero then the SFG can be redrawn as in Figure 7

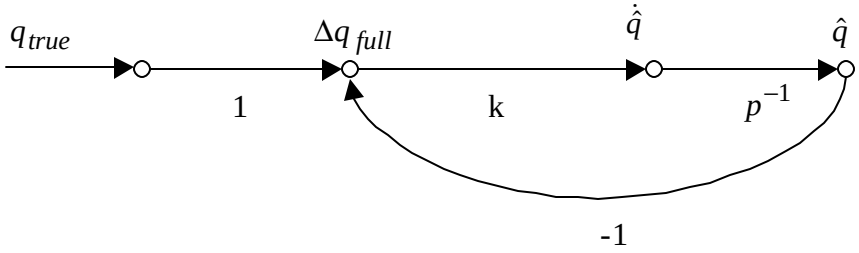

Figure 7: SFG for Initial Condition Error Analysis

From Eq. 4.23

$$\Delta q_{full} = (0\ -\partial_x\ -\partial_y\ -\partial_z) \qquad \text{(Eq. 4.28)}$$

Since the first component did not change, $\hat{q}$ will have the form

$$\hat{q} = (1\ \hat{x}\ \hat{y}\ \hat{z}) \qquad \text{(Eq. 4.29)}$$

Using Eq. 4.27, the transform domain SFG for the scalar $\hat{x}$ is as follows

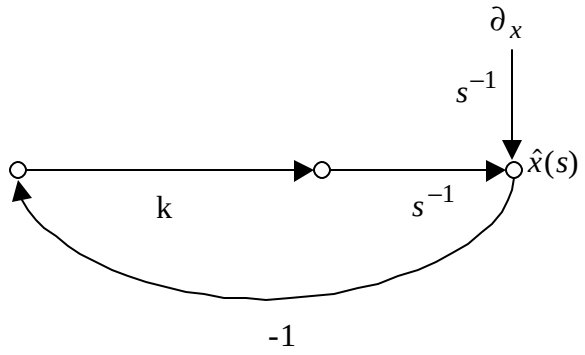

Figure 8: Transform Domain SFG for *$\hat{x}(s)$*

From this SFG, and application of Mason's formula it follows that [MCGH98B]

$$\frac{\hat{x}(s)}{\partial_x} = \frac{s^{-1}}{1 + ks^{-1}} = \frac{1}{s + k} \qquad \text{(Eq. 4.30)}$$

and therefore

$$\hat{x}(t) = \partial_x e^{-kt} \qquad \text{(Eq. 4.31)}$$

Equivalent results apply for $\hat{y}(t)$ and $\hat{z}(t)$. Thus, any transient errors in $\hat{q}$ resulting from erroneous input from the magnetometer and the accelerometer will persist for a time inversely proportional to k. Specifically if

$$\tau_{\Delta q} = \frac{1}{k} \qquad \text{(Eq. 4.32)}$$

then for any disturbance ∂_x, the resulting error in the x component of $\hat{q}$ will be

$$\varepsilon_{\hat{x}}(t) = \partial_x e^{\frac{-t}{\tau_{\Delta q}}} \qquad \text{(Eq. 4.33)}$$

This error will be reduced to 37% of the initial value by time $t = \tau_{\Delta q}$. This shows that the filter must use a big k value. On the other hand, if the maneuver time

constant is $\tau_{maneuver}$, it is required that $\tau_{\Delta q}$ be much larger than this value to suppress maneuver noise. Therefore, the qualitative requirement becomes

$$\frac{1}{\tau_{maneuver}} >> k >> \frac{1}{\tau_{bias}} \qquad \text{(Eq. 4.34)}$$

and the constraint for *?t* [MCGH98C] is

$$\Delta t < \frac{1}{k} \qquad \text{(Eq. 4.35)}$$

Reasonable values for Δt and k can be further adjusted by experimental means.

3. Rate Sensor Bias Correction

Drift is the tendency of sensor output to change over time with no change in sensor input and causes relentlessly increasing orientation measurement errors. This error is sometimes termed a bias. The integration of a bias-ridden angular rate signal will cause a steady build-up of error over time. This leads to an incorrect estimation of the body orientation relative to the Earth-fixed coordinate system. Angular rate sensor biases typically change unpredictably over time, making a simple, complete compensation impossible [ROBE97].

The filter takes the angular rate sensor readings and computes the bias-corrected output, B_w. To find the bias-corrected output, the filter must estimate the bias error and use this estimate to correct subsequent measurements. Discrete low pass filter theory provides a method for obtaining a rate bias estimate [KUO95].

The quaternion filter bias estimation is based on Shallow Water AUV Navigation System (SANS) experience [ROBE97]. The approach taken to deal with this problem in SANS was to estimate an initial value for the bias by averaging the rate sensor output

before beginning maneuvering and then tracking the time-varying drift with a very long time constant low pass filter [MCGH98D]. The signal flow graph for $\dot{q}_{measured}$ is given in Figure 9.

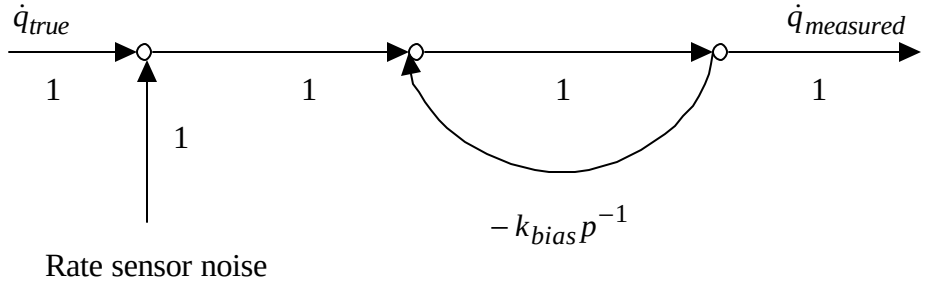

Figure 9: Bias Estimation Filter

Evidently

$$\frac{\dot{q}_{measured}}{\dot{q}_{true}} = \frac{1}{1 + k_{bias}p^{-1}} = \frac{p}{p + k_{bias}} \qquad \text{(Eq. 4.36)}$$

Thus, with the addition of bias estimation, the quaternion filter is no longer a complimentary filter since constant rates of rotation are eliminated. This effect can be minimized by applying the constraint

$$k >> k_{bias} \qquad \text{(Eq. 4.37)}$$

If k is too large, then the filter may become unstable or too much maneuver induced error may appear in $\hat{q}$ [MCGH98D].

The computed quaternion derivative, $\dot{q}$, is computed by the lower left-hand portion of the filter, Figure 10 [HENA97]. The expression for $\dot{q}$ is as given in Eq. 3.84.

$$\dot{q} = \frac{1}{2}\hat{q}^{B}w \qquad \text{(Eq. 4.38)}$$

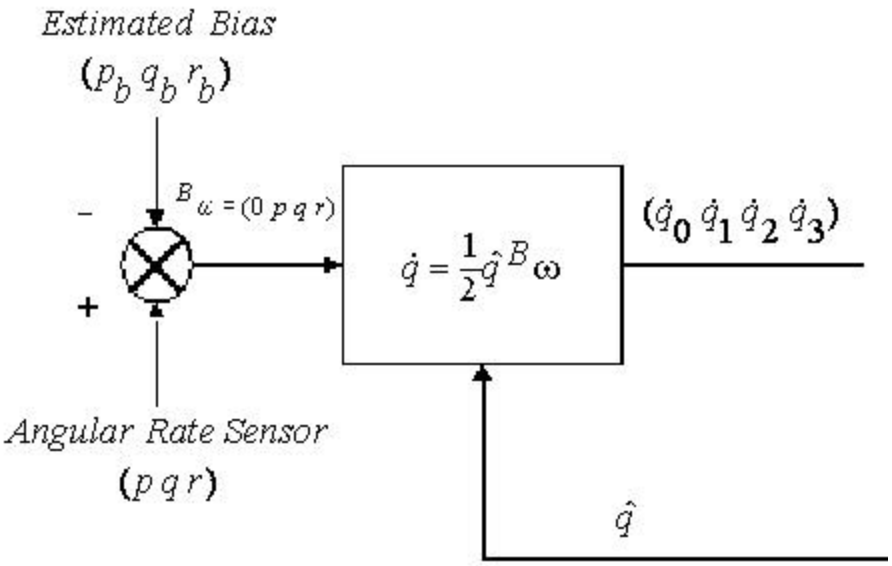

Figure 10: Lower Left Part of Quaternion Attitude Estimation Filter

The bias-corrected angular rate quaternion is corrected for the estimated bias (p_b q_b r_b). The corrected quaternion derivative $\dot{\hat{q}}$ is computed by

$$\dot{\hat{q}} = \dot{q} - \dot{q}_e \qquad \text{(Eq. 4.39)}$$

The corrected quaternion derivative is then numerically integrated and normalized. The resulting numerical integration equation becomes

$$\hat{q}_{n+1} = \hat{q}_n + \frac{1}{2}\hat{q}_n\,^B{}_w\,\Delta t + k\left[X^T X\right]^{-1} X^T e(\hat{q}_n)\Delta t \qquad \text{(Eq. 4.40)}$$

The normalized result is the estimated orientation quaternion, $\hat{q}$, which yields the next approximation. This approximated orientation quaternion can be used for the graphical representations of a tracked object in a virtual environment.

D. WHITE NOISE EFFECTS

White noise can be represented by a squence of statiscally independent numbers. The white noise effects on the filter for the Gauss-Newton method were tested by using a simulation program written in ANSI Common Lisp [HENA97]. A 6D white noise vector

was added to the measurement vector and a different white noise sample was used for every iteration. A scalar multiplier, a, was used to decrease the white noise effects on the filter. The modified filter with the white noise is shown in Figure 11.

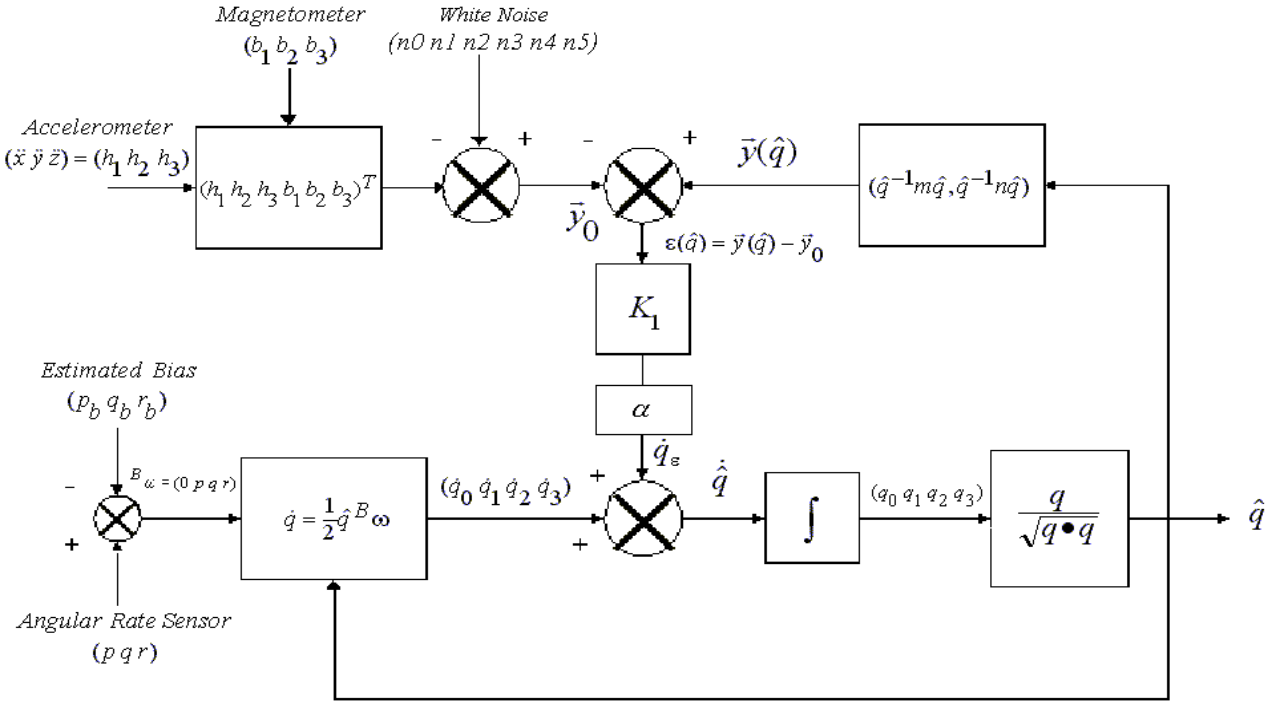

Figure 11: Modified Filter with White Noise

This modification is only made for testing the white noise effects on the filter; in real-time implementations this white noise must assumed to be in the measurements from the sensors. The filter responded to the simulated white noise in a corrective manner and the results were very close to the results with perfect noisless inputs.

Two important results were obtained from this test. The first one was that reducing a improves results in the long term, but takes longer to converge. The second result was that for tracking motion, the optimal a depends on the rate of motion and the noise level. The simulation results without white noise and with white noise are presented in Figure 12 and Figure 13 respectively. The Figure 12 shows an exponential

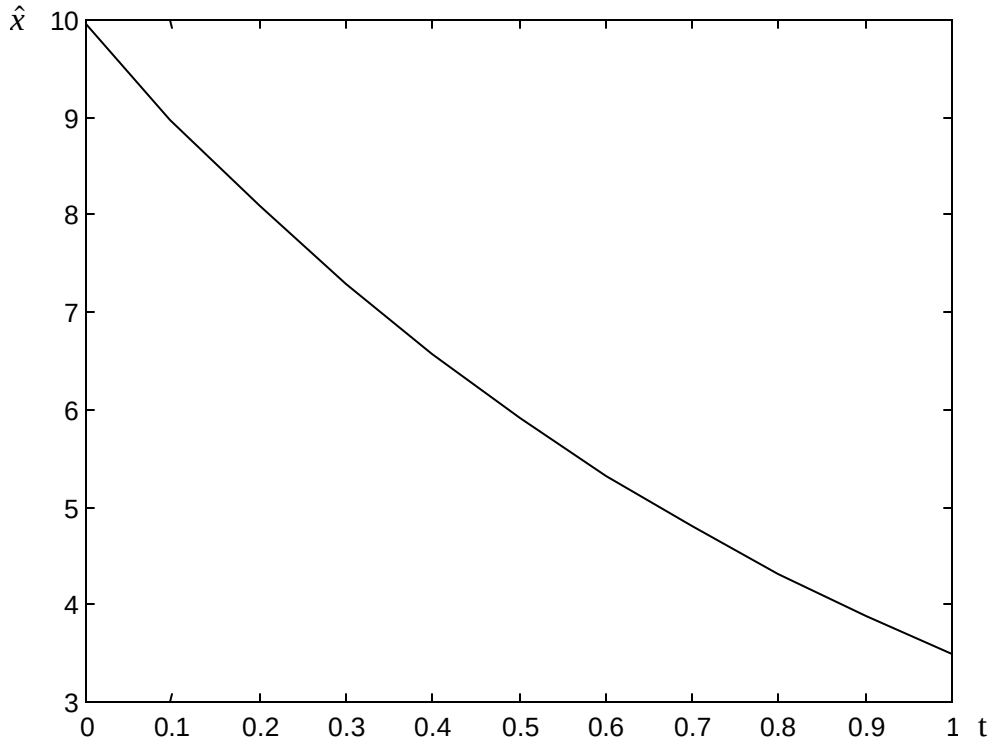

Figure 12: 10 Degree Offset, a=0.1, ?t=0.1, No Noise

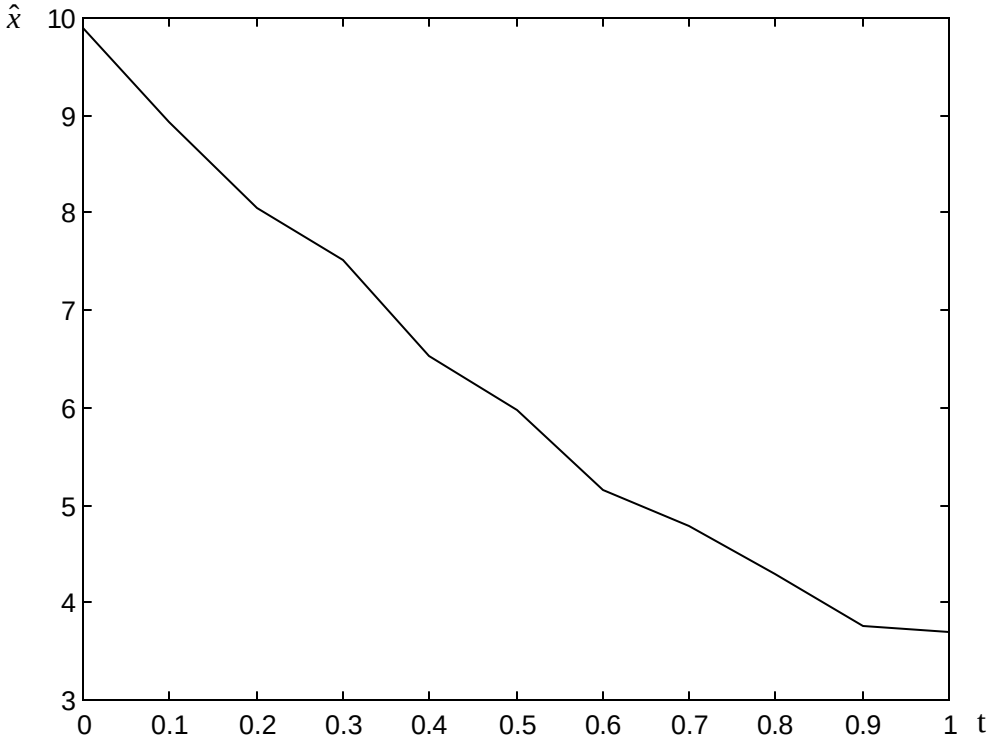

Figure 13: 10 Degree Offset, a=0.1, ?t=0.1, With Gaussian White Noise, Noise Standard Deviation=0.57 Degree

convergence. This result proves the value of the linearization theory and the error convergence formula given in Eq. 4.33.

E. SUMMARY

The Quaternion Attitude Estimation Filter offers significant improvements over filters using Euler angles. Inputs are obtained from a 3-axis accelerometer, a 3-axis angular rate sensor and a 3-axis magnetometer. No singularities or divide by zero errors exist for any orientation and no trigonometric functions are required. Using unit quaternions makes calculating inverses simple and provides an increase in computational efficiency over matrix multiplication.

The next chapter presents the hardware configuration of the quaternion attitude estimation filter system.

V. SYSTEM HARDWARE CONFIGURATION

A. INTRODUCTION

The hardware components of the quaternion attitude estimation filter system are configured for testing purposes. These components consist of a sensor block which holds a 3-axis magnetometer, a 3-axis accelerometer and a 3-axis rate sensor, a 12 VDC power source, an I/O connector, a 16-bit analog to digital (A/D) converter and a PC, Figure 14.

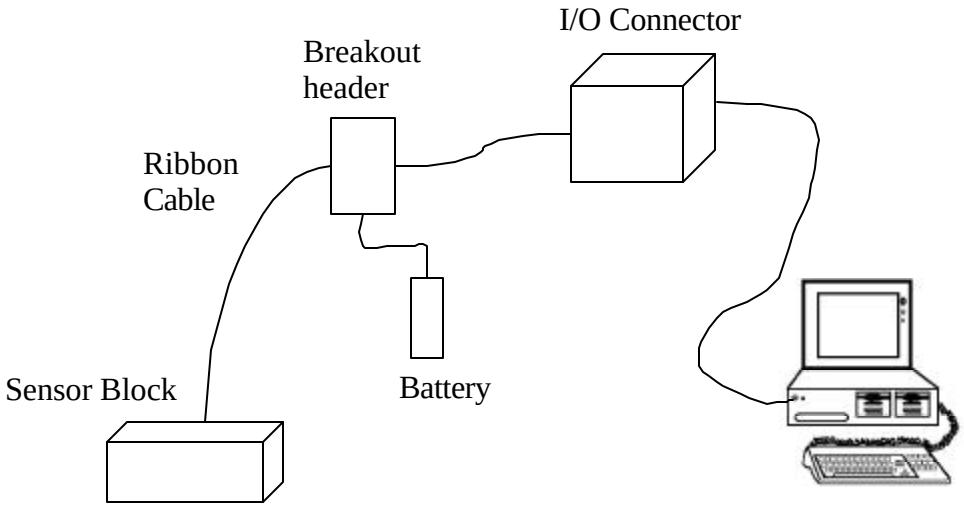

Figure 14: System Hardware Configuration

The power source supplies 12 VDC power for the sensor block. The ribbon cables, that carry actual output voltages from the sensors, are connected to the separate analog input channels on the I/O connector via a breakout header. The A/D converter card is physically in the PC and receives data from the sensors via the I/O connector. This chapter will summarize the hardware configuration and the main hardware component capabilities used in the quaternion attitude estimation filter system.

B. HARDWARE DESCRIPTION

1. Sensor Block

The sensor block contains three individual sensors. The main purpose of this sensor block is to implement a small 9-axis sensor system. The sensor block contains a 3-axis magnetometer, a 3-axis accelerometer, and a 3-axis rate sensor, Figure 15. This sensor block was constructed by [MCKI98].

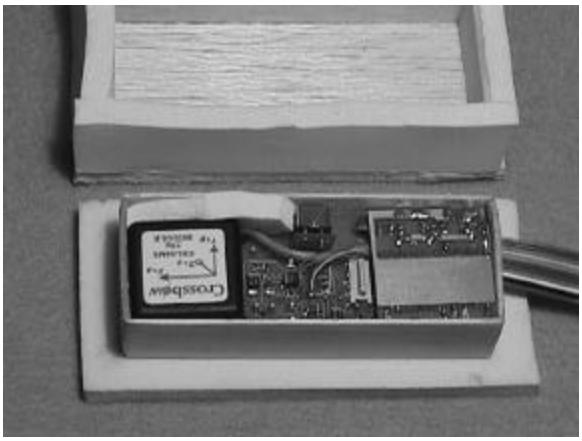

Figure 15: The Sensor Block [MCKI98]

The sensor block is covered with foam. The foam covering provides shock protection and a stable temperature environment for the sensors. The measurements of this package with foam coverage are 10.7 x 5 x 3.7 cm, and without foam coverage are 9.1 x 3.9 x 2 cm. The size of the sensor block is very small, but with smaller sensors, a smaller sensor block can be constructed.

The outputs from this sensor block are connected to the I/O connector by a ribbon cable via a breakout header. The DC power supply is also connected to this breakout header.

The 3-axis magnetometer is a Honeywell HMC2003 type magnetic sensor, Figure 16. The HMC2003 is a very small hybrid chip and can detect magnetic fields of less than

40 microgauss about three axes. Some characteristics of HMC2003 are given in Table 1 [HONE98].

Figure 16: Honeywell HMC2003 3-axis Magnetic Sensor Hybird [HONE98]

Characteristic	Range	Units
Supply Voltage	6 – 15	VDC
Field Range	-2 – 2	gauss
Output Voltage	0.5 – 4.5	V
Resolution	40	µ gauss
Bandwidth	1	KHz
Null Field Output	2.3 – 2.7	V

Table 1: 3-axis Magnetometer Specifications

The 3-axis accelerometer is a Crossbow Technology Inc. CXL04M3 type 3-axis accelerometer, Figure 17. The CXL04M3 can measure accelerations about three axes. Characteristics of CXL04M3 are given in Table 2 [CROS98].

Figure 17: Crossbow CXL04M3 3-axis Accelerometer [CROS98]

Characteristic	Range	Units
Supply Voltage	+ 5	VDC
Span Range	±4 ± 5%	G
Output Voltage	0 – 5	V
Resolution	5	mGrms
Bandwidth	DC-100 ± 5%	Hz
Sensitivity	500 ± 5%	mV/G
Zero G Output	2.5 ± 0.1	V

Table 2: 3-axis Accelerometer Specifications

The 3-axis rate sensor is a Tokin America Inc. CG16D0 type solid state rate sensor. The CG16D0 can measure the angular velocities about three axes. Some characteristics of CG16D0 are given in Table 3 [TOKI98].

Characteristic	Range	Units
Supply Voltage	+ 5	VDC
Detect Range	± 90	deg/sec
Output Voltage	0 – 5	V
Bandwidth	100	Hz
Sensitivity	1.1 ± 20%	mV/deg/sec
Reference Voltage Output	2.4 ± 10%	V

Table 3: 3-axis Angular Rate Sensor Specifications

The sensor block can support 100 Hz sampling rates. The magnetometer has a 1 KHz bandwidth.

2. A/D Converter

The A/D converter is a National Instruments Corporation® [NATI98] PCI-MIO-16XE-50 data acquisition (DAQ) card, Figure 18. It is physically inserted into the PCI slot of a PC motherboard. The PCI-MIO-16XE-50 is a 16-bit A/D converter, it has 16 single-ended or 8 double-ended analog input channels. Its maximum sampling rate is 20 K samples/sec. The recommended warm up time is 15 minutes.

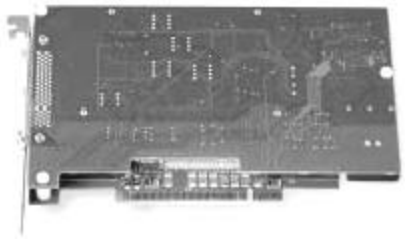

Figure 18: National Ins. PCI-MIO-16XE-50 Data Acquisition Card [NATI98]

It can measure analog input voltages between 0V – 10V (single sided) and ±10V (double sided). Sensor output voltage connections with A/D converter channels are shown in Table 4.

	Angular Rate Sensor			Accelerometer			Magnetometer		
	x	y	z	x	y	z	x	y	z
	p	q	r	h1	h2	h3	b1	b2	b3
Channel No	0	1	2	3	4	5	6	7	8
Pin No	68	33	65	30	28	60	25	57	34
Ground	67			29			64		

Table 4: Sensor Channel Connections

3. Other Components

The computer used in the research described in this thesis contains a 333 Mhz Intel Pentium II CPU and 64 MB of RAM. The operating system is Microsoft Windows 95™.

Figure 19: National Ins. SCB68 I/O Connection Board [NATI98]

The I/O connector is a National Instruments Corporation® [NATI98] SCB68 type I/O connection board, Figure 19. This I/O connector connects the output ribbon cable from the breakout header with the associated channels of the A/D converter.

C. SUMMARY

The hardware configuration described in this chapter was used for the quaternion attitude estimation filter system. For further information about the hardware components refer to the related product catalogs and the web sites of the manufacturers. The estimated equipment and material cost of the hardware except the PC is about $3.500. The next chapter presents the software design documentation of the quaternion attitude estimation filter.

VI. SOFTWARE DEVELOPMENT

A. INTRODUCTION

The Unified Modeling Language (UML) is a language for expressing the constructs and relationships of complex systems. A critical aspect of real-time systems is how time itself is handled. The design of a real-time system must identify the timing requirements of the system and ensure that the system performance is both correct and timely [DOUG98]. UML is particularly well suited for designing real-time embedded systems.

The features, concepts and visual notations used in UML are explained in [DOUG98]. During the software development process of the quaternion attitude estimation filter the Rational Rose 4.0 software design tool [RATI98] was used to implement the system in UML.

The steps in software development are system requirements analysis, system analysis, system design and implementation [DOUG98]. This chapter will present these steps for the quaternion attitude estimation filter software development.

B. SYSTEM REQUIREMENTS ANALYSIS

The quaternion attitude estimation filter (QAEF) is a complementary filter designed to track human limb segments through all orientations as part of an inertial tracking system. The QAEF uses three different sensors to obtain the information about the orientations of a tracked object. These sensors are a 3-axis accelerometer, a 3-axis angular rate sensor and a 3-axis magnetometer. The system gets these sensor readings through an I/O connector by using an analog to digital (A/D) converter. The system

samples and filters these data to produce an estimated orientation of the tracked object in the quaternion form.

The QAEF will filter the data from only one set of sensors, but is intended to be the building block for a body suit using several sensor blocks. The system use case diagram is presented in Figure 20.

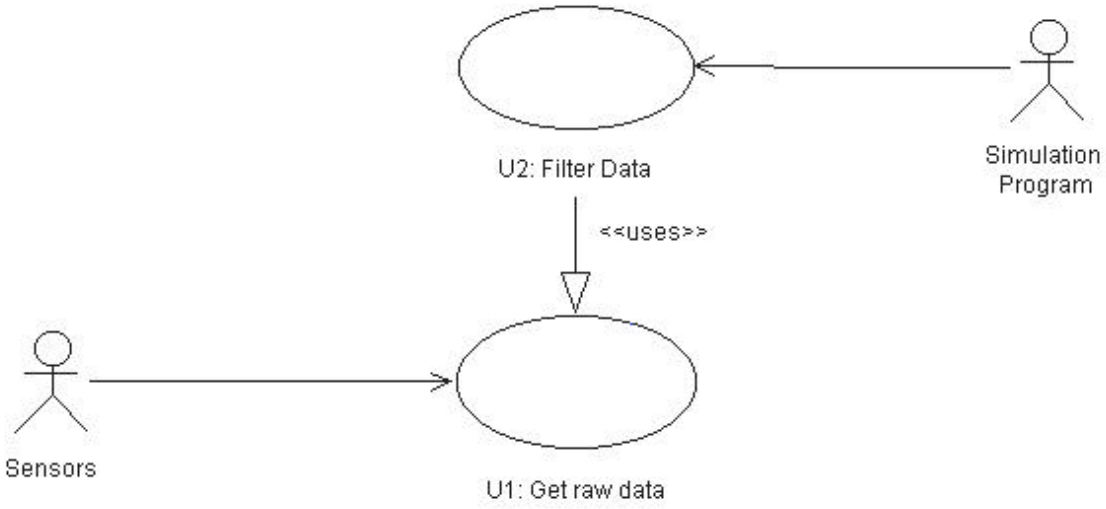

Figure 20: Use Case Diagram

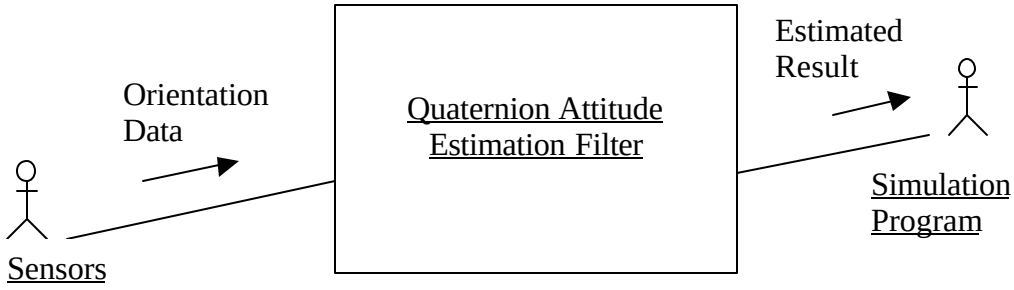

Figure 21: Context Diagram

The system's interaction with the actor objects is shown in the context diagram, Figure 21. A sequence diagram which shows the sequence of messages between objects is presented in Figure 22. The scenario diagram is the same as the sequence diagram,

because the system has a continuous behavior and there are no external events related to the system.

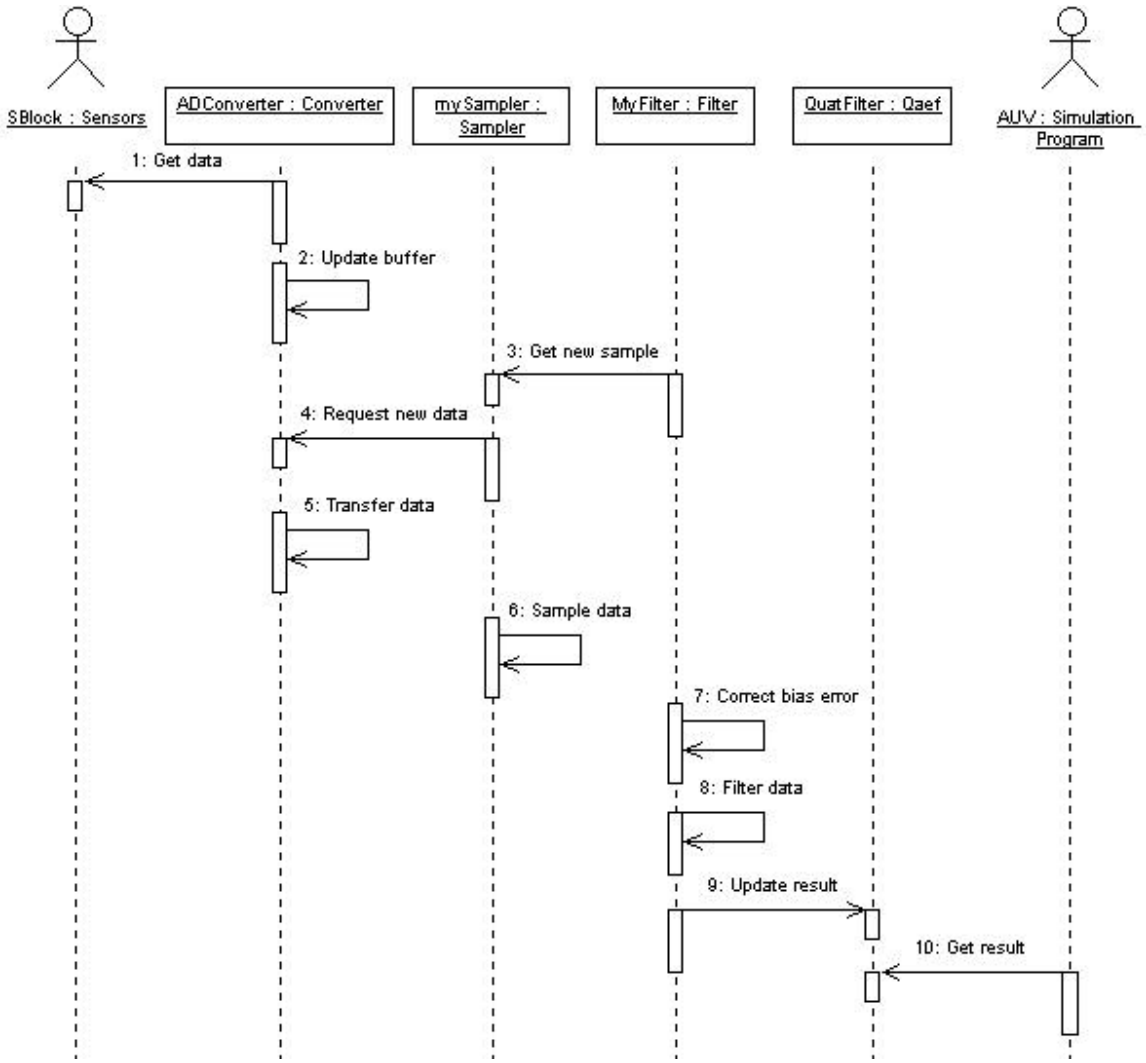

Figure 22: Sequence Diagram

The system will target a sampling rate between 40 Hz and 100 Hz, and an update rate of at least 30 Hz for real-time applications. The use cases and the system goals are presented in Table 5 and Table 6 respectively.

Goal 1	System has to get data from the sensors and convert it into digital data with a sampling rate of at least 40 Hz
Goal 2	System has to have at least a 30 Hz update rate at the end of the filtering process.

Table 5: System Goals

Use Case	(U1) Get raw data
Actor	Sensors
Goal	Traced to Goal 1
Preconditions	Sensors and A/D Converter are running
Description	Object rotates or keeps its rotation. System will always get the readings from the sensors, convert this data into digital data
Sub Use Case	None
Exception	None
Activities	Get the readings and write into a buffer
Postconditions	Continue getting data
Use Case	(U2) Filter data
Actor	Simulator program
Goal	Traced to Goal 2
Preconditions	Sensors, A/D Converter and system are running
Description	System will get the data from buffer and produce an estimated orientation in the quaternion form by filtering the data
Sub Use Case	None
Exception	None
Activities	Read data from the buffer, apply filtering algorithms and calculate an estimated orientation
Postconditions	Continue filtering processes

Table 6: System Use Cases

The major functions performed by the QAEF are data acquisition, data filtering, and updating, displaying and sending the estimated results. The actual system functions are given in Table 7.

System Functions	Rank	Use case
R1. Data Acquisition Functions		
R1.1 Get sensor readings	Essential	U1
R1.2 Write data into the buffer	Essential	U1
R1.3 Read data from the buffer	Essential	U1, U2
R2. Data Filtering Functions		
R2.1 Average the data	Essential	U2
R2.2 Calculate measurement vector	Essential	U2
R2.3 Calculate computed measurement vector	Essential	U2
R2.4 Calculate error	Essential	U2
R2.5 Calculate bias error	Essential	U2
R2.6 Calculate X-matrix	Essential	U2
R2.7 Calculate Gauss-Newton iteration	Essential	U2
R2.8 Calculate numerical integration	Essential	U2
R2.9 Convert data into digital	Essential	U2
R3. Matrix Operations		
R3.1 Multiply	Essential	U2
R3.2 Invert	Essential	U2
R3.3 Transpose	Essential	U2
R3.4 Convert to quaternion	Essential	U2
R3.5 Print	Essential	U2
R4. Quaternion Operations		
R4.1 Add	Essential	U2
R4.2 Subtract	Essential	U2

Table 7: System Functions

R4.3 Quaternion multiplication	Essential	U2
R4.4 Rotation	Essential	U2
R4.5 Dot product	Essential	U2
R4.6 Derivative	Essential	U2
R4.7 Normalize	Essential	U2
R4.8 Print	Essential	U2
R5. Output Functions		
R5.1 Print to the screen	Desired	U2
R5.2 Write into a file	Optional	U2
R5.3 Write into a buffer	Optional	U2

Table 7: System Functions (Cont'd)

C. SYSTEM ANALYSIS

The QAEF system boundary is limited by the A/D converter. The actual filter application will pass the estimated results to a 3D simulation or to a tracking program. For this research the system will print the results to the screen and will write them into a file. The system behavior is continuous and will implement a Proportional Integral Derivative (PID) control loop. The system will run on a PC under the Microsoft Windows 95™ operating system. The identified objects are shown in the object diagram, Figure 23. The system state chart showing the general system behavior is in presented in Figure 24.

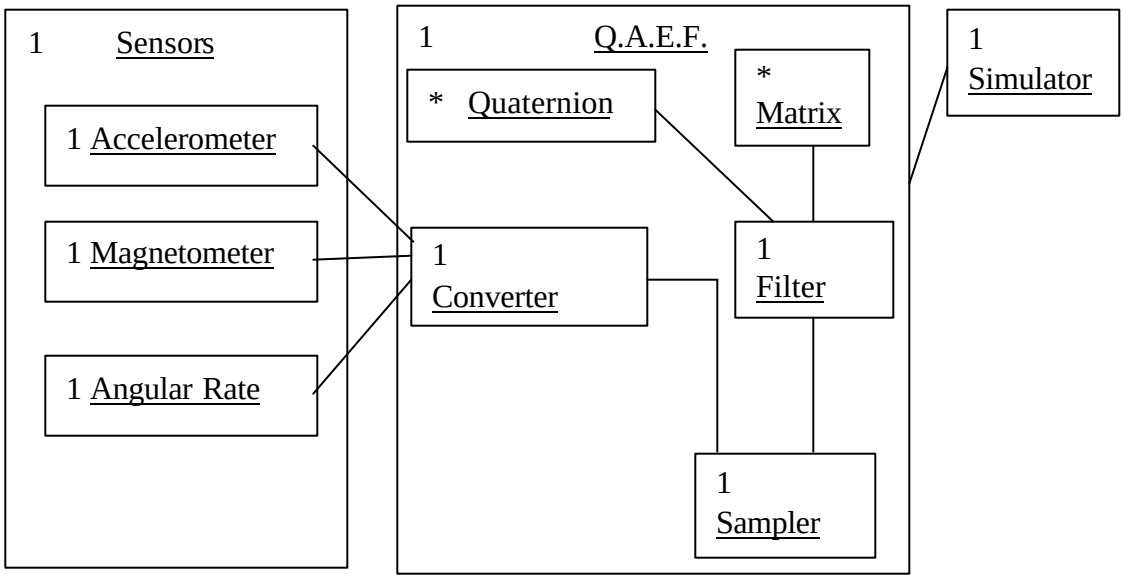

Figure 23: Object Diagram

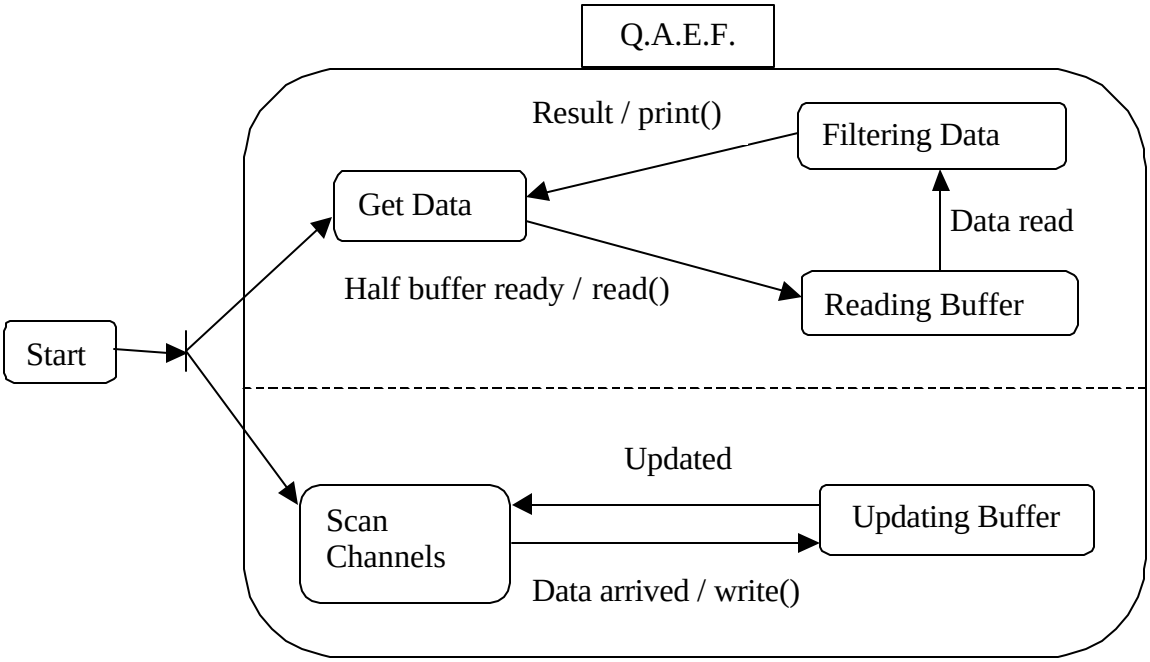

Figure 24: System Statechart

The system functions are distributed to the identified classes. The associations and the relations of these classes are determined from the object diagram and from the sequence diagram to create the class diagram for the system. The class diagram is presented in Figure 25.

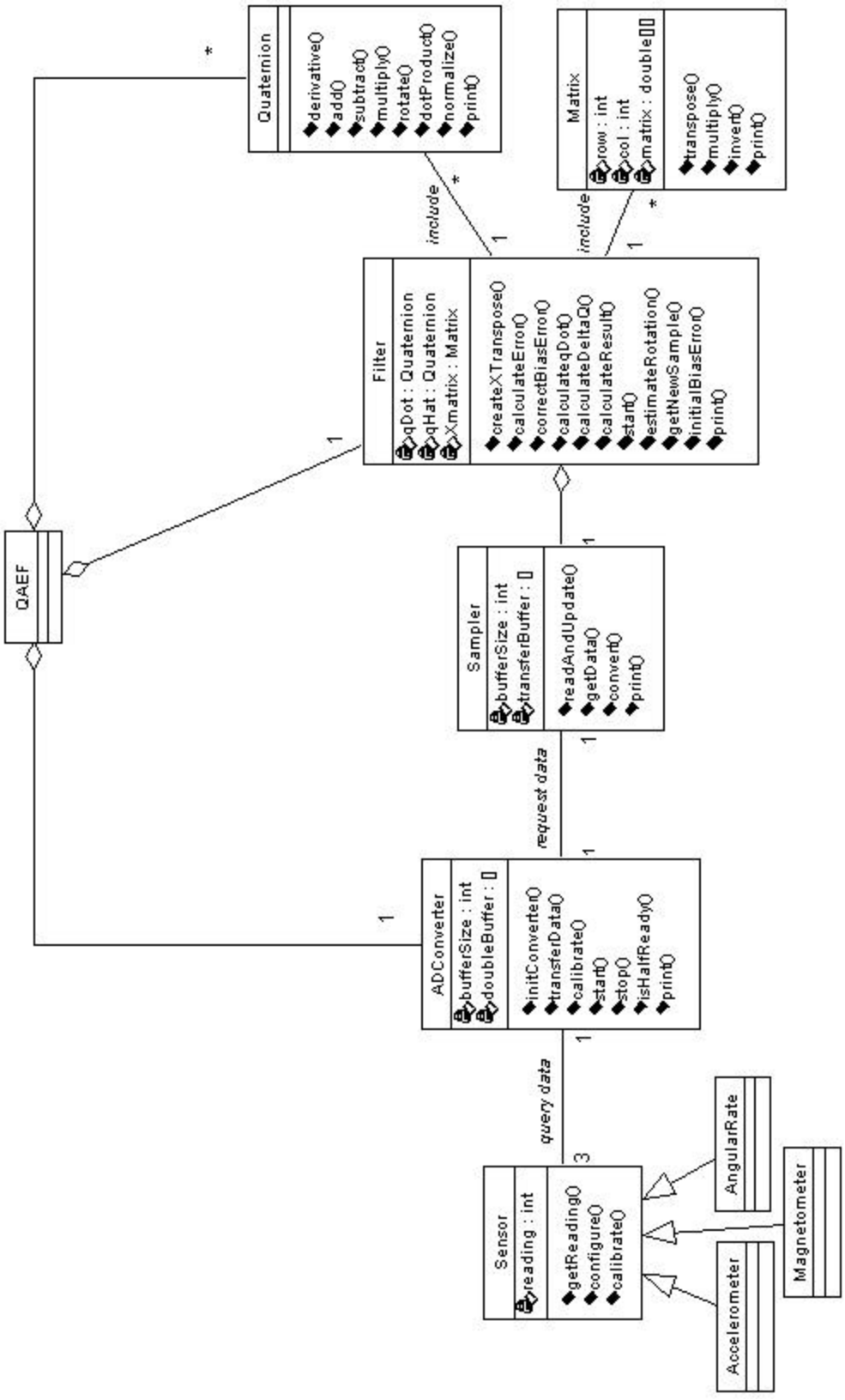

Figure 25: Class Diagram

D. SYSTEM DESIGN

Design details the largest scale software structures, such as subsystems, packages and tasks, and specifies the internal primitive data structures and algorithms within individual classes [DOUG98]. The QAEF system hardware configuration is explained in Chapter V. The deployment diagram for the QAEF hardware is presented in Figure 26.

The tasks are identified as “Get Sensor Readings”, “Sample Data” and “Filter Data”. The “Get Sensor Readings” task gets sensor readings and writes these readings into the double buffer. The Filter Data task calls its server task, the Sample Data task, to get the current sample readings. The Filter Data task receives these samples via an array and then filters the data to produce an estimated output result. Sample Data task sends a data transfer request signal to the Get Sensor Readings task, and then the Get Sensor Readings task updates the transfer buffer with half of the double buffer. The Filter Data task should wait for buffer updating and the sampling data processes. The data transferring process from the double buffer to the transfer buffer will have a timeout. During this period, the Filter Data task will be waiting for the sampled data and for the Get Sensor Readings task to finish this transfer. If for some reason this process takes too much time, then the Filter Data task should continue the data filtering process with the current updated data. The length of the timeout is changeable and is dependent on the speed of the filtering and the acquisition processes. The timeout value is calculated to be 1/60 of a second, but it can be changed during the system tests.

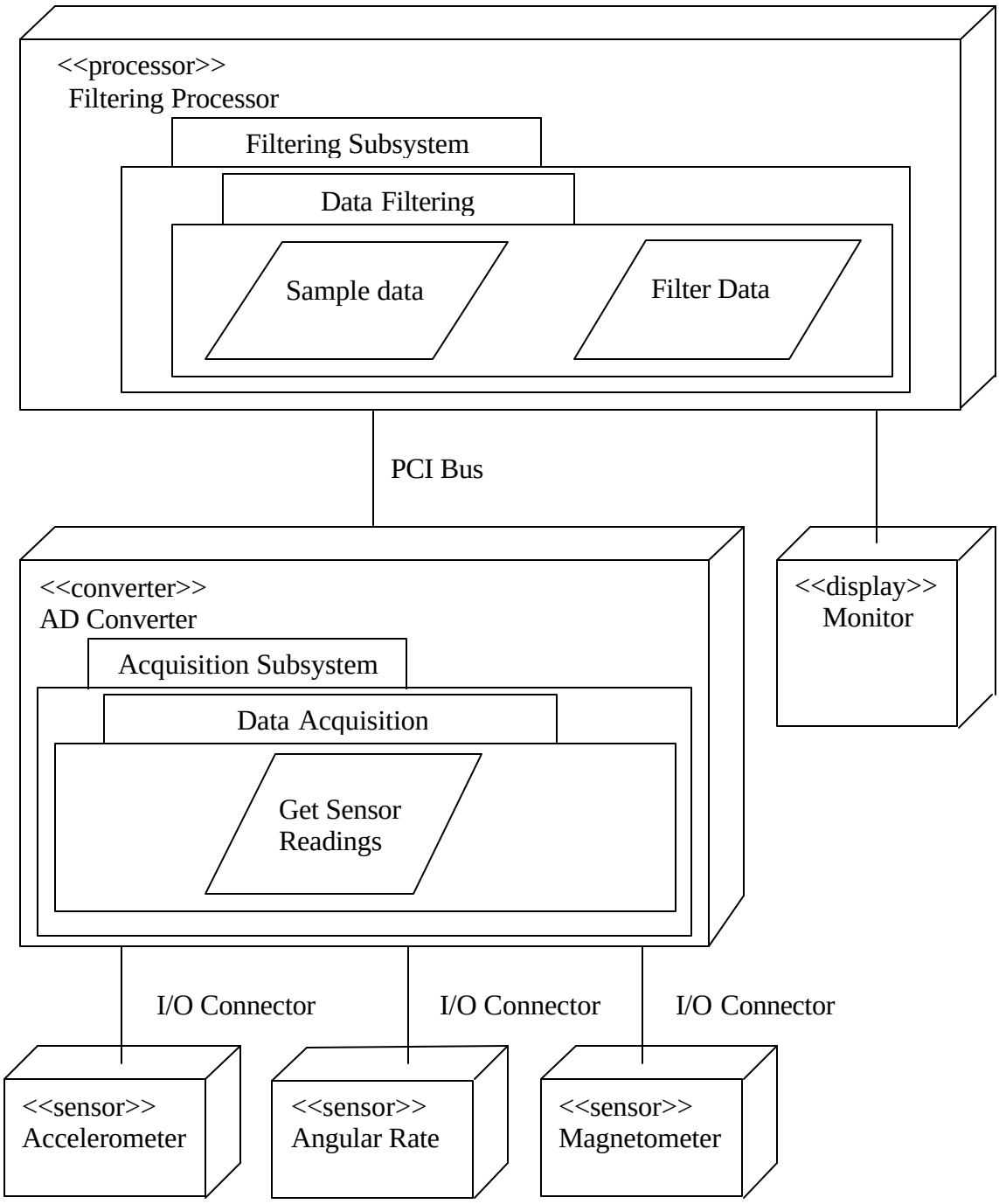

Figure 26: System Deployment Diagram

The QAEF system has two subsystems namely the Filtering Subsystem and the Acquisition Subsystem. The Filtering Subsystem is responsible for data sampling and for

data filtering. This subsystem is composed of the *Filter*, *Sampler*, *Quaternion*, and *Matrix* classes in the Data Filtering package, Figure 27.

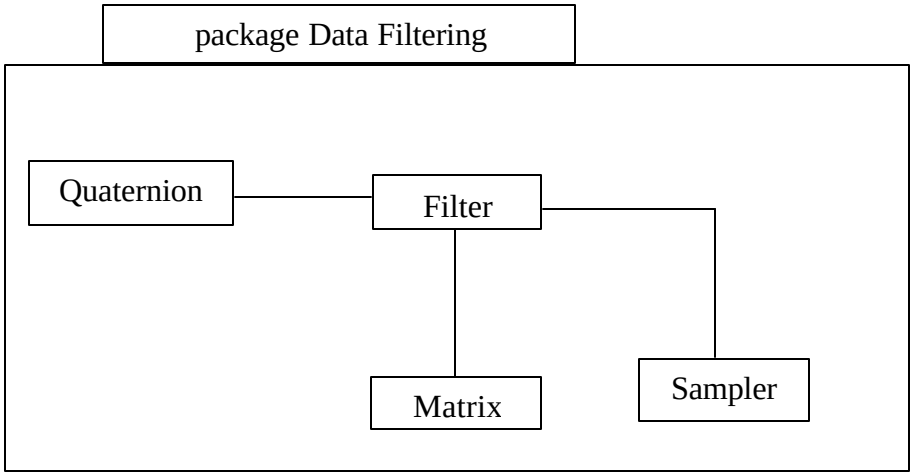

Figure 27: Filtering Subsystem Structure

The active objects in Data Filtering package are the Filter and Sampler. The operations of these active objects in the Filtering Subsystem are shown in Figure 28. The Filter is responsible for the calculations in the filtering process. The Filter object requests data from the Sampler. Upon receiving the sampled data, it calculates the bias correction (the Filter object must calculate the initial bias correction for the first step only). After applying this bias error to the sampled data, it starts the calculation process to find an estimated result. The result will be printed on the screen and will be written into a file. The Filter object state model is presented in Figure 29.

The Sampler is responsible for initializing the calibration values, getting data from the converter, averaging this data, and providing this sampled data to the Filter along with the time interval of the samples. The calibration values are the zero level voltages for the sensor readings. These values can either be found by the Sampler or can

be hard coded. The Sampler object gets data from the converter when requested by the Filter. The Sampler samples the data by averaging, and then sends the samples

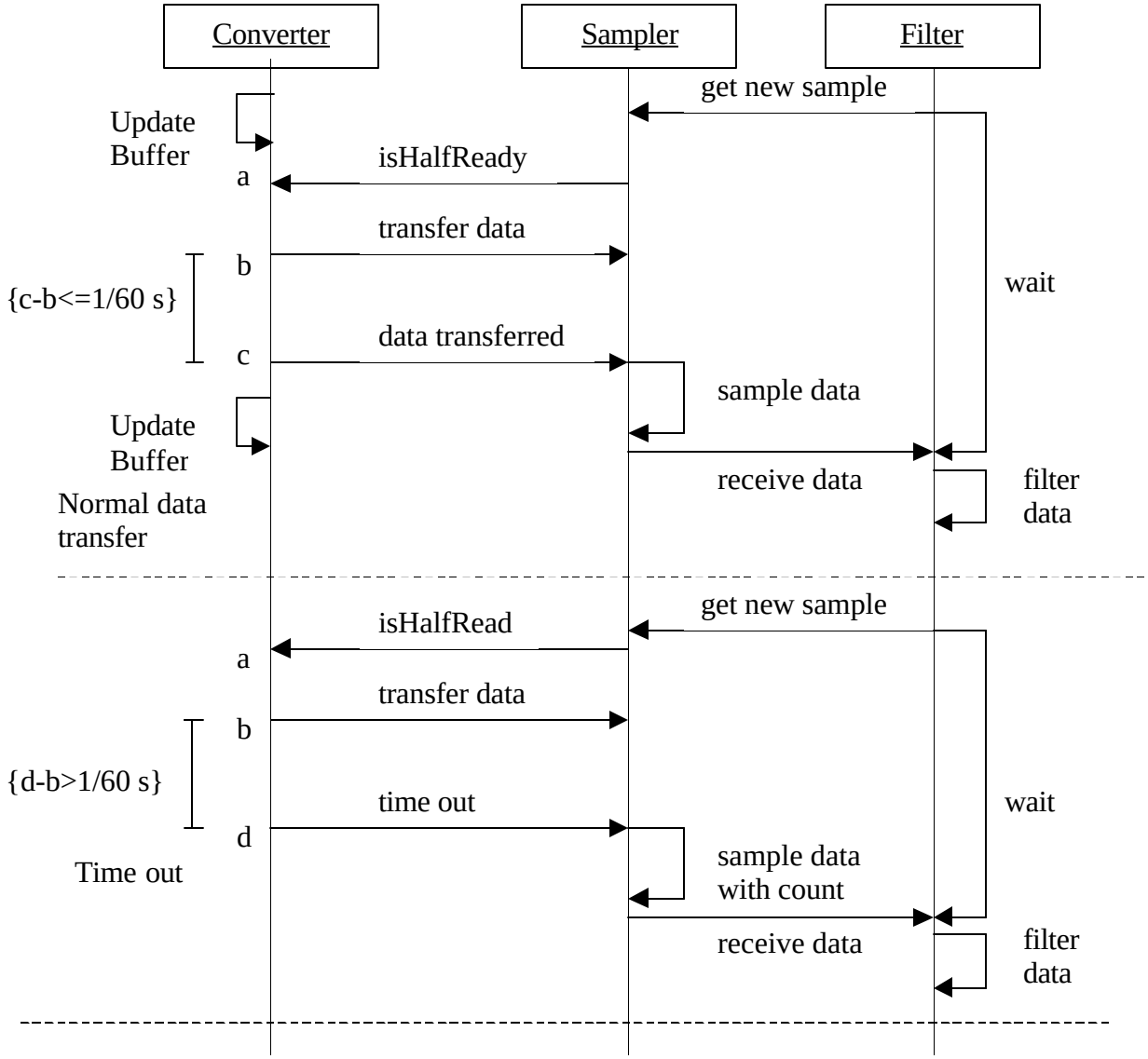

Figure 28: Filtering Subsystem Operations

and the time interval to the Filter. The Sampler will wait for the data transmission from the double buffer into the transfer buffer. If this transmission exceeds the transmission time, then a timeout occurs. The state model for the Sampler is presented in Figure 30.

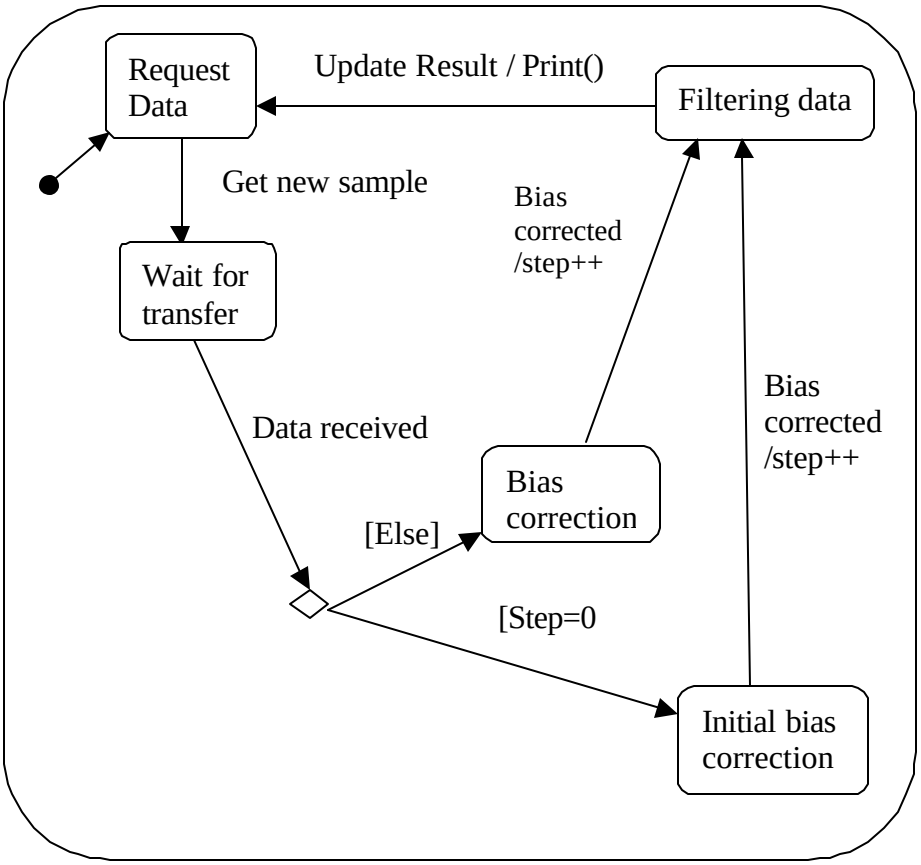

Figure 29: Filter Object State Model

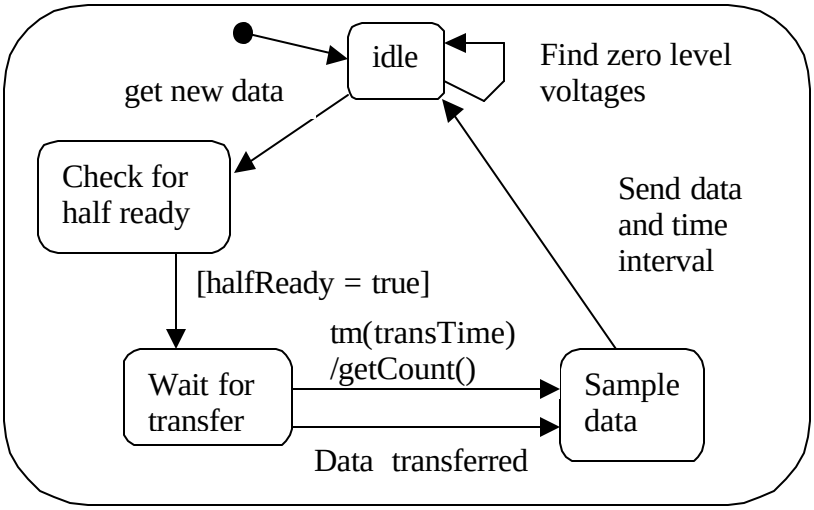

Figure 30: Sampler Object State Model

The Quaternion class supports the system with quaternion operations and with quaternion arithmetic as explained in Chapter III. This class could support any simulation or tracking program using quaternions.

The Matrix class supports the system with necessary matrix operations. These operations are used in the filtering process and are special for this design. It also has the ability to make combinations of and conversions between quaternions and matrices.

The Acquisition Subsystem is responsible for getting data from the sensors and for updating the double buffer with this data. This subsystem is composed of the *Converter* class in the Data Acquisition package, as depicted in Figure 31.

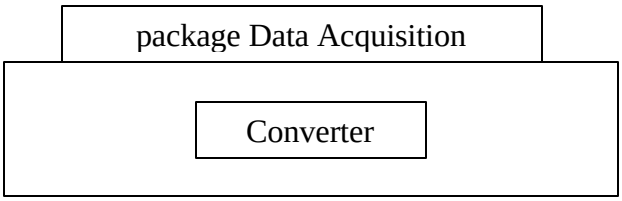

Figure 31: Acquisition Subsystem Structure

The Converter is an active object. The Converter class provides an interface between the system and the converter hardware. The Converter is responsible for initiating and calibrating the A/D converter, for monitoring the data collection process, for updating the double buffer with new data, and for transferring halves of the double buffer into the transfer buffer. The Converter communicates with the Sampler during this transfer process. It checks for the timeout condition and monitors all events for any possible error in data acquisition, such as buffer overflow and buffer overwrite. The Converter concurrently checks the channels for new data, and checks for any requests from the Sampler. The operations of the Converter in the Data Acquisition Subsystem are shown in Figure 32.

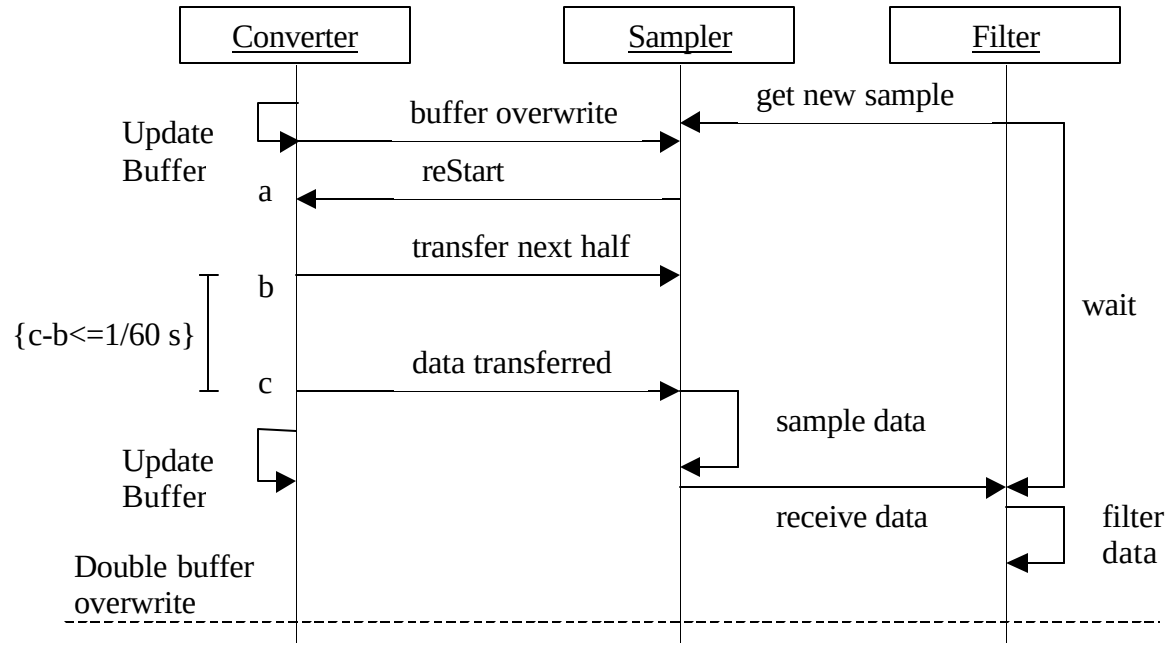

Figure 32: Acquisition Subsystem Operations

The Converter updates the double buffer when new data is received and it transfers the data into the transfer buffer. The state model for the Converter is presented in Figure 33.

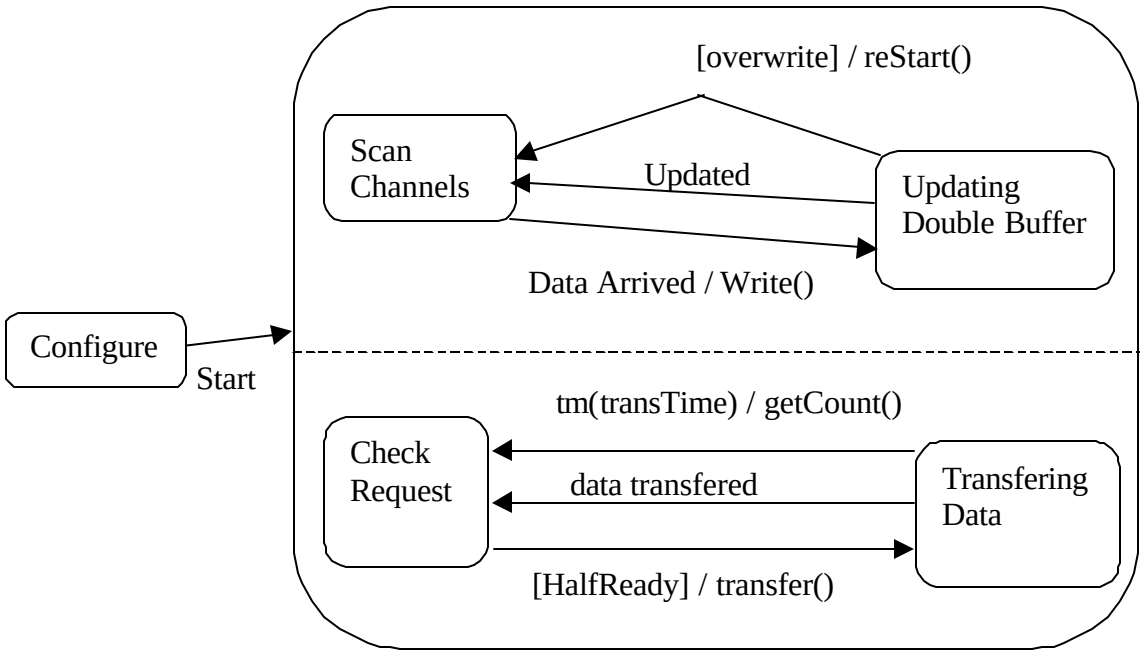

Figure 33: Converter Object State Model

E. SYSTEM IMPLEMENTATION

The system was implemented using the C++ programming language and the Visual C++ 5.0 compiler. The attribute and the variable names given are the same as those used to define the Quaternion Attitude Estimation Filter in Chapter IV. The attributes and the variables are commented and the functions are explained along with their parameters in the function headers. Terminology consistency between the documentation and the code were strived for throughout implementation.

The Gauss-Newton method and the filtering equations are implemented in the Filter class as they are presented in Chapter IV. Operator overloading is used in both the Quaternion class and the Matrix class to clarify the source code and to provide for ease of use. The Quaternion class has the ability to perform every quaternion operation explained in Chapter III. Every class has the ability to print its pre-specified attributes by overloading the stream insertion operator.

Utility functions to write into and read from a file are provided for the user. All classes have default constructors, but the user has the opportunity to initialize the objects by reading the initial values from a configuration file.

A function library, nidaq32.lib, is used to program the converter and these functions are explained with their parameters in [NATI98A]. The converter has two types of data acquisition, single buffered and double buffered. Single buffered operations are relatively simple to implement, and can usually take advantage of the full hardware speed of data acquisition device. However, more sophisticated applications involving larger amounts of data input at higher rates require more advanced techniques. One such

technique is double buffering. This allows continuous, uninterrupted input of large amounts of data [NATI98B]. In this implementation, double buffering was chosen.

The converter is configured to work in bipolar mode, ±10 V, with board gain 2. In this configuration, the digital ranges for the actual voltages will be between –32768 and 32767. These values are used to initialize the converter object.

F. SUMMARY

The QAEF software is designed to track motion in all orientations in real time. The UML and its visual notations were used to analyze and to design it. The software is written in C++, with the Visual C++ 5.0 compiler, for use on an IBM-compatible processor. This chapter presented a general overview of the software development process. The source code is presented in Appendix C. The test plans and the actual testing of a system are the final steps in the software development process. The next chapter will present the testing methodology and the test results of the QAEF system.

VII. SYSTEM TESTING

A. INTRODUCTION

System testing is the final step in software and hardware development. The quantitative tests focused on the software and the accuracy of the system. This chapter presents system calibration, testing methodology and the test results.

B. SYSTEM CALIBRATION

To initialize the system three values must be provided to the QAEF object, namely converter number, buffer depth and a constant gain value for *k*. The converter number specifies the converter to use when there is more than one converter. The buffer depth indirectly determines update rate. Up to the point of buffer overwrite errors, small buffer depths will provide faster update rates. The constant gain value *k* is discussed in Chapter IV. Throughout the quantitative tests, *k* was set to 3 and a was kept less than 1.

Most of the calibration is done within the *sampler* object. The *sampler* object has a self-calibrating function namely *findZeroVoltages*. This function finds the zero level voltages for angular rate sensors and accelerometers. The sensor block must be sitting on a level surface for several milliseconds to find these zero level voltages. Hard coded values may also be used. These hard coded values can be determined by taking the average values of the maximum and the minimum readings of each sensor in the sensor block. The self-calibration process can be enabled or disabled. When the self-calibration is enabled the *findZeroVoltages* function will initialize the *sampler*, otherwise the *sampler* will initialize itself with hard coded values. In actual tests, initializing the Earth magnetic field to the first reading from magnetometer obtained during calibration provided better results and simplified the initialization process.

All calibration of the A/D converter is done automatically by the *converter* object. The parameters to calibrate the A/D converter are chosen to get the best performance from the A/D converter. These parameters can easily be changed using the parameters discussed in [NATI98A] and making changes to the *converter* object.

C. TESTING METHODOLOGY

The sensor block was mounted on the tilt table using a non-ferrous extension, which has approximately the length of a human forearm. The tilt table was programmed to rotate 45 degrees at a rate of 10 degrees per second on every axis for a series of roll, pitch and yaw tests. After initializing the system, three rotations were completed by the tilt table in each test. During each test, a 15 to 20 second stabilization period followed each movement. The scale factors and buffer depth values were adjusted to obtain the desired update rate and accuracy.

D. TEST RESULTS

The following results were obtained using the hardware and software described in Chapters V and VI. To achieve an update rate of 55 Hz, the graphical representation of orientation was disabled. The update rate with the graphical orientation representation and a single processor was approximately 15 Hz. The roll, pitch, and yaw test results are presented in Figures 34, 35, and 36 respectively.

Estimated orientation was within 1 degree of actual steady orientation throughout the tests. The smoothness of the graphs indicates excellent dynamic response. The small impulses observed at the starting points of motion were hypothesized to be linear acceleration effects exaggerated by a whipping motion of the non-ferrous extension block. It is expected that adjusting the filter scale factors and gain values will reduce this

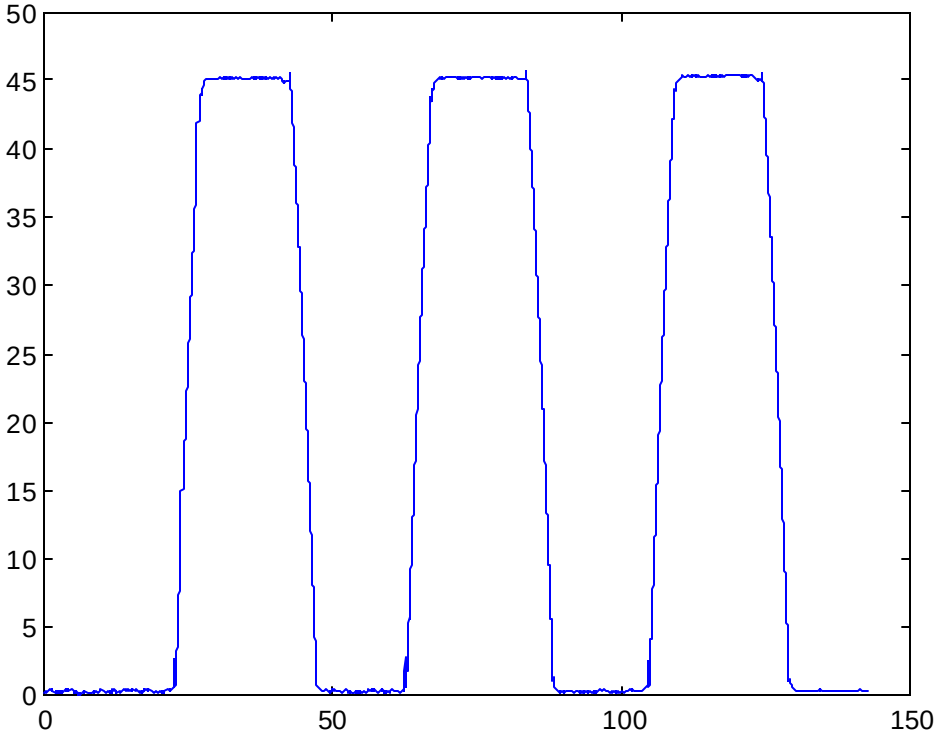

Figure 34: 45 Degree Roll Test, 10 deg/sec, a = 0.054, 55 Hz.

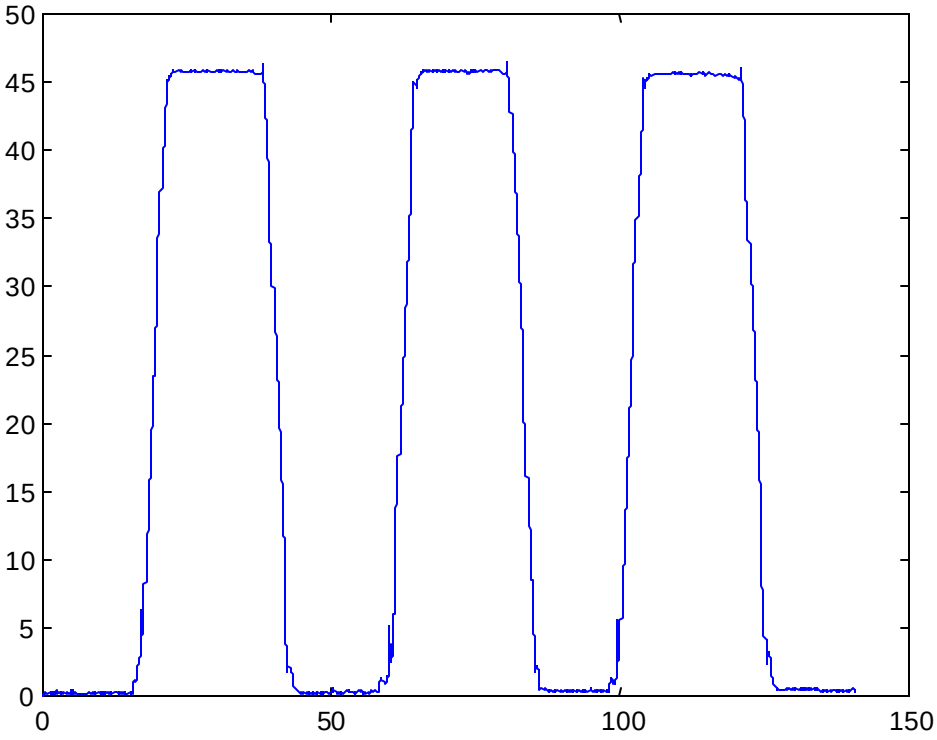

Figure 35: 45 Degree Pitch Test, 10 deg/sec, a = 0.054, 55 Hz.

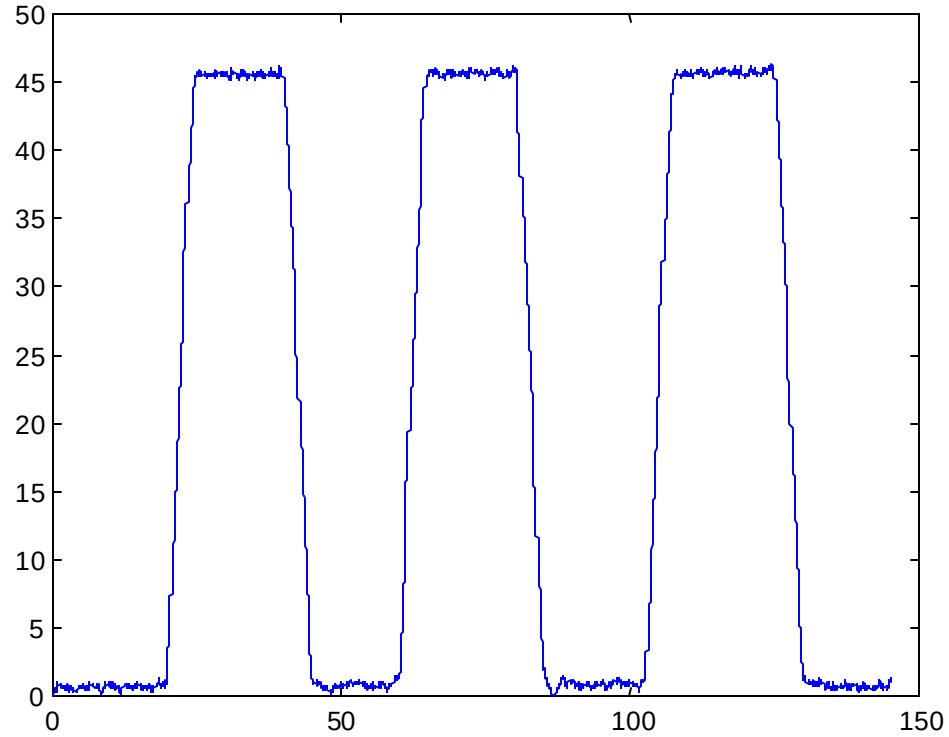

Figure 36: 45 Degree Yaw Test, 10 deg/sec, a = 0.054, 55 Hz.

effect and improve the overall accuracy and dynamic response. The transition times observed in the plots are around 4.5-5 seconds as expected for a 10-degree per second rotation rate to 45 degrees. In qualitative tests, the system was able to track all orientations, including those in which pitch equaled 90 degrees; the same orientations normally cause singularities in Euler angle filters. The qualitative tests also show that the system could easily be combined with a simulation program and track motion in real time.

E. SUMMARY

This chapter has described the system calibration, the system testing methodology and the qualitative and quantitative dynamic tilt table test results. These tests show that the system's lag is very low, and the accuracy is within 1 degree when a proper calibration is completed. The prototype system update rate can be adjusted between 15

and 55 Hz. It appears to be capable of tracking human motion in real time. The final chapter of this thesis will review this research, reach some conclusions and make recommendations for future work.

VIII. SUMMARY AND CONCLUSIONS

A. SUMMARY

Several techniques exist to track human motion and many research efforts have addressed body tracking systems and orientation filters. The most commonly used method for defining orientation is Euler angles. Motion tracking systems using Euler angles are not capable of tracking objects in all orientations due to gimbal lock singularities. An alternative method, the use of unit quaternions, has been gaining popularity in the graphics community since mid 80's. A quaternion based attitude estimation filter has been proposed to overcome these singularities. This thesis described the development and testing of a prototype inertial tracking system based on a quaternion attitude estimation filter. Test results indicate that human motions could be efficiently tracked with this system.

B. CONCLUSIONS

This research continues that begun in [HENA97]. The presented system is the first inertial body tracking system based on quaternions [DURL95, SKOP96]. Using unit quaternions prevented the singularities and increased computational efficiency.

The system resolution is very good, and accuracy is within 1 degree. Robustness and registration are very good in a homogenous environment, but relatively strong magnetic fields will effect the system. Since the system is sourceless, it can be assumed that there is no limit to sociability in a very large homogenous environment. The prototype sensor is small and it is light. This makes the system user friendly and easy to use. Smaller sensor blocks, wristwatch size, could be produced with smaller sensors in

the near future. The system was constructed using low cost, off the shelf components. System cost and size would be reduced with mass production

C. RECOMMENDATIONS FOR FUTURE WORK

The system will be used as a part of the on going body suit project [ZYDA97] and could also be used in the NPS AUV project [YUN97]. The system and multiple sensor blocks can be combined with the quaternion human model [USTA99] for real time human motion tracking applications. Using multiple sensor blocks will decrease the system performance on a single processor system. To prevent this, a multiprocessor architecture should be developed for full human body motion tracking applications. Tracking humans in large environments will require a wireless tracking system. Developing a wireless tracking system architecture should be considered.

The system has a self-calibrating module or the scalar numbers can be hard coded for system calibration. An improved calibration procedure should be developed to achieve better results. The filter can be used with other types of sensors. Integrating better sensors into the system should be considered.

APPENDIX A. DERIVATION OF X MATRIX

The X matrix is defined by (Eq. 4.13) as

$$X_{ij}^{T} = \left[\frac{\partial y_i}{\partial \hat{q}_j} \right]_{4x6} \qquad \text{(Eq. A.1)}$$

The elements of the 4x6 matrix come from the partial derivatives of the components of the computed measurement vector, $\vec{y}(\hat{q})$, given by (Eq. 4.8)

$$\vec{y}(\hat{q}) = (\hat{q}^{-1} m \hat{q}, \hat{q}^{-1} n \hat{q}) \qquad \text{(Eq. A.2)}$$

and as given in (Eq. 4.6) and (Eq. 4.7)

$$\hat{h} = \hat{q}^{-1} m \hat{q} \qquad \text{(Eq. A.3)}$$

$$\hat{b} = \hat{q}^{-1} n \hat{q} \qquad \text{(Eq. A.4)}$$

Therefore, taking the partial derivative of this equation with respect to $\hat{q}_0$, the result is

$$\frac{\partial \vec{y}}{\partial \hat{q}_0} = \frac{\partial}{\partial \hat{q}_0} (\hat{q}^{-1} m \hat{q}, \hat{q}^{-1} n \hat{q}) \qquad \text{(Eq. A.5)}$$

Applying the product rule to (Eq. A.5), it follows that

$$\frac{\partial \vec{y}}{\partial \hat{q}_0} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_0} m \hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_0}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_0} n \hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_0} \right) \qquad \text{(Eq. A.6)}$$

where

$$\frac{\partial \hat{q}}{\partial \hat{q}_0} = (1\ 0\ 0\ 0) \qquad \text{(Eq. A.7)}$$

and

$$\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_0} = (1\ 0\ 0\ 0) \qquad \text{(Eq. A.8)}$$

Likewise, for $q_1, q_2,$ and q_3;

$$\frac{\partial \vec{y}}{\partial \hat{q}_1} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_1} m\hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_1}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_1} n\hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_1} \right) \qquad \text{(Eq. A.9)}$$

$$\frac{\partial \vec{y}}{\partial \hat{q}_2} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_2} m\hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_2}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_2} n\hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_2} \right) \qquad \text{(Eq. A.10)}$$

$$\frac{\partial \vec{y}}{\partial \hat{q}_3} = \left(\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_3} m\hat{q} + \hat{q}^{-1} m \frac{\partial \hat{q}}{\partial \hat{q}_3}, \frac{\partial \hat{q}^{-1}}{\partial \hat{q}_3} n\hat{q} + \hat{q}^{-1} n \frac{\partial \hat{q}}{\partial \hat{q}_3} \right) \qquad \text{(Eq. A.11)}$$

where the corresponding quaternion partial derivatives are

$$\frac{\partial \hat{q}}{\partial \hat{q}_1} = (0\ 1\ 0\ 0) \qquad \text{(Eq. A.12)}$$

$$\frac{\partial \hat{q}}{\partial \hat{q}_2} = (0\ 0\ 1\ 0) \qquad \text{(Eq. A.13)}$$

$$\frac{\partial \hat{q}}{\partial \hat{q}_3} = (0\ 0\ 0\ 1) \qquad \text{(Eq. A.14)}$$

and, the partial derivatives of the inverse are given by

$$\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_1} = (0\ -1\ 0\ 0) \qquad \text{(Eq. A.15)}$$

$$\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_2} = (0\ 0\ -1\ 0) \qquad \text{(Eq. A.16)}$$

$$\frac{\partial \hat{q}^{-1}}{\partial \hat{q}_3} = (0\ 0\ 0\ -1) \qquad \text{(Eq. A.17)}$$

The partial derivatives as defined in (Eq. A.6), (Eq. A.9), (Eq. A.10), and (Eq. A.11) result in the partial derivatives of the m and n vectors with respect to $\hat{q}_0, \hat{q}_1, \hat{q}_2,$ and $\hat{q}_3,$ respectively.

Taking the results computed above, the X matrix can be constructed as follows

$$X = \left[\frac{\partial \vec{y}}{\partial \hat{q}_0} \;\; \frac{\partial \vec{y}}{\partial \hat{q}_1} \;\; \frac{\partial \vec{y}}{\partial \hat{q}_2} \;\; \frac{\partial \vec{y}}{\partial \hat{q}_3} \right]_{6x4} \qquad \text{(Eq. A.18)}$$

Note that the partial derivatives are column vectors in the X matrix, and that the transpose of X is required when used in the filter. Thus, (Eq. A.18) becomes [HENA97]

$$X^T = \begin{bmatrix} \frac{\partial \vec{y}}{\partial \hat{q}_0} \\ \frac{\partial \vec{y}}{\partial \hat{q}_1} \\ \frac{\partial \vec{y}}{\partial \hat{q}_2} \\ \frac{\partial \vec{y}}{\partial \hat{q}_3} \end{bmatrix}_{4x6} \qquad \text{(Eq. A.19)}$$

Then the transpose of the X matrix is

$$X^T = \begin{bmatrix} \frac{\partial \hat{h}_1}{\partial \hat{q}_0} & \frac{\partial \hat{h}_2}{\partial \hat{q}_0} & \frac{\partial \hat{h}_3}{\partial \hat{q}_0} & \frac{\partial \hat{b}_1}{\partial \hat{q}_0} & \frac{\partial \hat{b}_2}{\partial \hat{q}_0} & \frac{\partial \hat{b}_3}{\partial \hat{q}_0} \\ \frac{\partial \hat{h}_1}{\partial \hat{q}_1} & \frac{\partial \hat{h}_2}{\partial \hat{q}_1} & \frac{\partial \hat{h}_3}{\partial \hat{q}_1} & \frac{\partial \hat{b}_1}{\partial \hat{q}_1} & \frac{\partial \hat{b}_2}{\partial \hat{q}_1} & \frac{\partial \hat{b}_3}{\partial \hat{q}_1} \\ \frac{\partial \hat{h}_1}{\partial \hat{q}_2} & \frac{\partial \hat{h}_2}{\partial \hat{q}_2} & \frac{\partial \hat{h}_3}{\partial \hat{q}_2} & \frac{\partial \hat{b}_1}{\partial \hat{q}_2} & \frac{\partial \hat{b}_2}{\partial \hat{q}_2} & \frac{\partial \hat{b}_3}{\partial \hat{q}_2} \\ \frac{\partial \hat{h}_1}{\partial \hat{q}_3} & \frac{\partial \hat{h}_2}{\partial \hat{q}_3} & \frac{\partial \hat{h}_3}{\partial \hat{q}_3} & \frac{\partial \hat{b}_1}{\partial \hat{q}_3} & \frac{\partial \hat{b}_2}{\partial \hat{q}_3} & \frac{\partial \hat{b}_3}{\partial \hat{q}_3} \end{bmatrix} \qquad \text{(Eq. A.20)}$$

APPENDIX B. ..... DERIVATION OF GRADIENT OF THE ERROR CRITERION FUNCTION

In the filter, the gradient of the error criterion function $f(\hat{q})$, is given by (Eq.4.17)

$$\nabla f(\hat{q}) = \left(\frac{\partial f}{\partial \hat{q}_0} \frac{\partial f}{\partial \hat{q}_1} \frac{\partial f}{\partial \hat{q}_2} \frac{\partial f}{\partial \hat{q}_3} \right)^T = 2X^T{}_{4x6} e(\hat{q})_{6x1} \qquad \text{(Eq. B.1)}$$

where $f(\hat{q})$, is given by (Eq.4.14)

$$f(\hat{q})_{1x1} = e(\hat{q})^T{}_{1x6} e(\hat{q})_{6x1} \qquad \text{(Eq. B.2)}$$

In order to simplify the derivation of the gradient, consider the the case of one dimensional orientation vectors, where the measured vector is $\vec{y}_0 = (h_1)$ and the calculated vector is $\vec{y}(\hat{q}) = (\hat{h}_1)$. The error then becomes

$$e(\hat{q}) = \hat{h}_1 - h_1 \qquad \text{(Eq. B.3)}$$

and

$$f(\hat{q}) = e^2 = (\hat{h}_1 - h_1)^2 = \hat{h}_1^2 - 2h_1\hat{h}_1 + h_1^2 \qquad \text{(Eq. B.4)}$$

Taking the partial derivative of (Eq.B.4) with respect to $\hat{q}_0$ yields

$$\frac{\partial f}{\partial \hat{q}_0} = 2\hat{h}_1 \frac{\partial \hat{h}_1}{\partial \hat{q}_0} - 2h_1 \frac{\partial \hat{h}_1}{\partial \hat{q}_0} = 2\frac{\partial \hat{h}_1}{\partial \hat{q}_0}(\hat{h}_1 - h_1) = 2\frac{\partial \hat{h}_1}{\partial \hat{q}_0} e(\hat{q}) \qquad \text{(Eq. B.5)}$$

Likewise, for the partial derivatives with respect to $\hat{q}_1, \hat{q}_2$, and $\hat{q}_3$ [HENA97]. Thus, extending this result to 6 dimensional case and arranging the partial derivatives in matrix form yields the derivation of the gradient of the error criterion function.

The modeling error can be written

$$e(\hat{q}) = \begin{bmatrix} \hat{h}_1 - h_1 \\ \hat{h}_2 - h_2 \\ \hat{h}_3 - h_3 \\ \hat{b}_1 - b_1 \\ \hat{b}_2 - b_2 \\ \hat{b}_3 - b_3 \end{bmatrix} \quad \text{(Eq. B.6)}$$

Based on this error, $f(\hat{q})$ becomes

$$f(\hat{q}) = (\hat{h}_1 - h_1)^2 + (\hat{h}_2 - h_2)^2 + (\hat{h}_3 - h_3)^2 + (\hat{b}_1 - b_1)^2 + (\hat{b}_2 - b_2)^2 + (\hat{b}_3 - b_3)^2 \quad \text{(Eq. B.7)}$$

Taking the partial derivative of (Eq. B.7) with respect to $\hat{q}_0$

$$\frac{\partial f}{\partial \hat{q}_0} = \frac{\partial}{\partial \hat{q}_0}\left((\hat{h}_1 - h_1)^2 + (\hat{h}_2 - h_2)^2 + (\hat{h}_3 - h_3)^2 + (\hat{b}_1 - b_1)^2 + (\hat{b}_2 - b_2)^2 + (\hat{b}_3 - b_3)^2\right) \quad \text{(Eq. B.8)}$$

we obtain

$$\frac{\partial f}{\partial \hat{q}_0} = 2\begin{bmatrix} \frac{\partial \hat{h}_1}{\partial \hat{q}_0} & \frac{\partial \hat{h}_2}{\partial \hat{q}_0} & \frac{\partial \hat{h}_3}{\partial \hat{q}_0} & \frac{\partial \hat{b}_1}{\partial \hat{q}_0} & \frac{\partial \hat{b}_2}{\partial \hat{q}_0} & \frac{\partial \hat{b}_3}{\partial \hat{q}_0} \end{bmatrix} \begin{bmatrix} \hat{h}_1 - h_1 \\ \hat{h}_2 - h_2 \\ \hat{h}_3 - h_3 \\ \hat{b}_1 - b_1 \\ \hat{b}_2 - b_2 \\ \hat{b}_3 - b_3 \end{bmatrix} \quad \text{(Eq. B.9)}$$

Rows two through four are produced by taking the partial derivative of $f(\hat{q})$ with respect to $\hat{q}_1, \hat{q}_2$, and $\hat{q}_3$.

Thus,

$$\nabla f(\hat{q}) = 2\begin{bmatrix} \frac{\partial \hat{h}_1}{\partial \hat{q}_0} & \frac{\partial \hat{h}_2}{\partial \hat{q}_0} & \frac{\partial \hat{h}_3}{\partial \hat{q}_0} & \frac{\partial \hat{b}_1}{\partial \hat{q}_0} & \frac{\partial \hat{b}_2}{\partial \hat{q}_0} & \frac{\partial \hat{b}_3}{\partial \hat{q}_0} \\ \frac{\partial \hat{h}_1}{\partial \hat{q}_1} & \frac{\partial \hat{h}_2}{\partial \hat{q}_1} & \frac{\partial \hat{h}_3}{\partial \hat{q}_1} & \frac{\partial \hat{b}_1}{\partial \hat{q}_1} & \frac{\partial \hat{b}_2}{\partial \hat{q}_1} & \frac{\partial \hat{b}_3}{\partial \hat{q}_1} \\ \frac{\partial \hat{h}_1}{\partial \hat{q}_2} & \frac{\partial \hat{h}_2}{\partial \hat{q}_2} & \frac{\partial \hat{h}_3}{\partial \hat{q}_2} & \frac{\partial \hat{b}_1}{\partial \hat{q}_2} & \frac{\partial \hat{b}_2}{\partial \hat{q}_2} & \frac{\partial \hat{b}_3}{\partial \hat{q}_2} \\ \frac{\partial \hat{h}_1}{\partial \hat{q}_3} & \frac{\partial \hat{h}_2}{\partial \hat{q}_3} & \frac{\partial \hat{h}_3}{\partial \hat{q}_3} & \frac{\partial \hat{b}_1}{\partial \hat{q}_3} & \frac{\partial \hat{b}_2}{\partial \hat{q}_3} & \frac{\partial \hat{b}_3}{\partial \hat{q}_3} \end{bmatrix} \begin{bmatrix} \hat{h}_1 - h_1 \\ \hat{h}_2 - h_2 \\ \hat{h}_3 - h_3 \\ \hat{b}_1 - b_1 \\ \hat{b}_2 - b_2 \\ \hat{b}_3 - b_3 \end{bmatrix} \qquad \text{(Eq. B.10)}$$

which can be rewritten [BACH97A]

$$\nabla f(\hat{q}) = 2X^T e(\hat{q}) \qquad \text{(Eq. B.11)}$$

where the transpose of the X matrix is as defined in Appendix A.

APPENDIX C. SOURCE CODE

A. QAEF.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Qaef.h
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Header file for Qaef
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __QAEF_H__
#define __QAEF_H__

#include <iostream.h>
#include <iomanip.h>
#include "converter.h"
#include "filter.h"
#include "quaternion.h"


class Qaef{

public:

      // default constructor
      Qaef (int converterNo ,unsigned int transferBufSize = 400,
              unsigned int kValue = 150);

      // destructor
      ~Qaef();

      // starts the filtering process
      double start();
```

```
        // stops the converter
        void stop();

        // returns the estimated quaternion
        Quaternion getResult();

private:

        // AD converter
        Converter * qConverter;

        // Filter
        Filter * qFilter;

};

#endif

// end of file Qaef.h
```

B. QAEF.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Qaef.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Qaef
// - Initializes A/D converter and filter
// - Returns estimated result
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <stdio.h>
#include <stdlib.h>
#include "qaef.h"
#include "converter.h"
#include "filter.h"
#include "quaternion.h"


//---------------------------------------------------------------------
// Function:      Qaef()
// Return Value:  None
// Parameters:    None
// Description:   Default constructor
//---------------------------------------------------------------------
Qaef::Qaef(int deviceNo, unsigned int bufSize, unsigned int k )
:qConverter (new Converter(deviceNo,bufSize)),
qFilter (new Filter(qConverter, k))
{
}//end Constructor
```

```
//---------------------------------------------------------------------
// Function:      ~Qaef()
// Return Value:  None
// Parameters:    None
// Description:   Destructor
//---------------------------------------------------------------------
Qaef::~Qaef()
{
      qConverter->~Converter();
      qFilter->~Filter();
      cerr<<"QAEF deleted"<<endl;
}// end destructor


//---------------------------------------------------------------------
// Function:      start()
// Return Value:  None
// Parameters:    None
// Description:   Starts filtering process
//---------------------------------------------------------------------
double Qaef::start()
{

      qConverter->start();

      return qFilter->start();
}//end start()


//---------------------------------------------------------------------
// Function:      getResult()
// Return Value:  Quaternion - estimated result
// Parameters:    None
// Description:   Returns the estimated result
//---------------------------------------------------------------------
Quaternion Qaef::getResult()
{
      return qFilter->estimateRotation();
}// end getResult()


//---------------------------------------------------------------------
// Function:      stop()
// Return Value:  None
// Parameters:    None
// Description:   Stops the converter
//---------------------------------------------------------------------
void Qaef::stop()
{
      qConverter->stop();
      return;
}// end stop

//end of file Qaef.cpp
```

C. FILTER.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Filter.h
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Header file for Filter.cpp
// - Filters data
// - Estimates an result
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __FILTER_H__
#define __FILTER_H__

#include <iostream.h>
#include <iomanip.h>
#include "matrix.h"
#include "quaternion.h"
#include "sampler.h"
#include "converter.h"


class Filter{

      // overloaded operator<<
      friend ostream &operator<<(ostream &,const Filter &);

public:

      // default constructor
      Filter (Converter *, unsigned int k = 5);

      // destructor
      ~Filter();

      // copy constructor
      Filter (const Filter &);
```

```
        // starts filtering process
        double start();

        // estimates a new rotation
        Quaternion estimateRotation();

private:

        Sampler *mySampler;  // my server sampler

        Quaternion n,     // Earth's unit magnetic field
              m,          // Earth's unit gravity
              h,          // local vertical from accelerometer
              b,          // local magnetic field from magnetometer
              bw,         // rate , from angular rate sensor
              deltaq,     // Gauss Newton iteration
              qDot,       // derivation of angular rate
              hHat, bHat, // in computed measurement vector
              qHat;       // estimated quaternion

        // partial derivatives
        const Quaternion QHATZERO,
              QHATONE, ONEINV,
              QHATTWO, TWOINV,
              QHATTHREE, THREEINV;

        Matrix x,               // X matrix
              xTranspose,       // X transpose matrix
              errorqHat,        // error q hat
              errorTranspose,   // error transpose
              temp,temp1;       // temp matrices for calculations

        int tau;                //for bias error

        bool selfCalibration;  // for Sampler to find zero level voltages

        double sampleWeight,    // for bias error
              biasWeight,
              deltaT,           // time between two sampling in seconds
              * biasError,      // bias errors for angular rates
              * samples,        // received readings from sampler
              kValue,           // k value, will be a constant
              errorMagnetitude; // error's magnetitude

        // gets new sampled data from sampler
        void getNewSample();

        // calculates initial bias error
        void initialBiasError();

        // corrects bias error by low pass filtering
        void correctBiasError();

        // sets the quaternions with new data
        void setElements();
```

```
        // calculates error
        void calculateError();

        // creates X Transpose matrix
        void createXTranspose();

        // calculates delta q quaternion
        void calculateDeltaQ();

        // calculates qDot quaternion
        void calculateqDot();

        // calculates final result
        void calculateResult();

        // converts the final matrix solution in to quaternion
        void convertToDeltaq(Matrix &);

        // writes results into a file
        void logResults();
};

#endif

// end of file Filter.h
```

D. FILTER.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Filter.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Filter
// - Filters data
// - Estimates an result
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <math.h>
#include <string.h>
#include <assert.h>
#include <sys/timeb.h>
#include <time.h>
#include <fstream.h>
#include <stdlib.h>
#include "filter.h"
#include "matrix.h"
#include "quaternion.h"
#include "sampler.h"
#include "util.h"

ofstream fresults;

//--------------------------------------------------------------------
// Function:      Filter()
// Return Value:  None
// Parameters:    None
// Description:   Default constructor
//--------------------------------------------------------------------
Filter::Filter (Converter * con, unsigned int kVal)
:n(Quaternion ("n", 0 , 0 , 60 , -15)),//for Monterey,CA,in Euler form
m(Quaternion ("m",0.0,0.0,0.0,-1.0)),
h(Quaternion ("h")),
```

```
b(Quaternion ("b")),
bw(Quaternion("bw")),
deltaq(Quaternion ("delta q")),
qDot(Quaternion ("q dot")),
hHat(Quaternion ("h hat")),
bHat(Quaternion ("b hat")),
qHat(Quaternion ("Result",1.0,0.0,0.0,0.0)),
QHATZERO(Quaternion("qhat zero",1.0,0.0,0.0,0.0)),
QHATONE(Quaternion ("qhat one",0.0,1.0,0.0,0.0)),
ONEINV(Quaternion ("qhat one inv",0.0,-1.0,0.0,0.0)),
QHATTWO(Quaternion ("qhat two",0.0,0.0,1.0,0.0)),
TWOINV(Quaternion ("qhat two inv",0.0,0.0,-1.0,0.0)),
QHATTHREE(Quaternion ("qhat three",0.0,0.0,0.0,1.0)),
THREEINV(Quaternion ("qhat three inv",0.0,0.0,0.0,-1.0)),
x(Matrix ("X",6,4)),
xTranspose(Matrix ("X Transpose",4,6)),
errorqHat (Matrix ("error q hat",6,1)),
errorTranspose(Matrix ("Error Transpose",1,6)),
errorMagnetitude(0.0),
temp (Matrix ("temp",4,4)),
temp1 (Matrix ("temp1",4,1)),
tau(100), deltaT(0.1),kValue(kVal),
sampleWeight(0.0), biasWeight(0.0),
selfCalibration(true) // sampler will find zero level voltages
{

      writeFile(fresults,"FilterResults.dat");

      mySampler = new Sampler(con);

      biasError = new double [3];
      assert (biasError !=0);

      samples = new double [9];
      assert (samples !=0);

      for (int i=0;i<3;i++){
            biasError [i] = 0.0;
      }
      for (int k=0;k<9;k++){
            samples [k] = 0.0;
      }

      qHat.normalize();

      // find the Earth's unit magnetic field , n
      Quaternion tt("tt",0,1,0,0); // just a temp quaternion
      n=n.toQuaternion();  // convert n to Quaternion
      n=n.rotation(tt);    // rotate n with (0 1 0 0)
      //cerr<<n<<endl;
      qHat.normalize();
      // will be set to b
      n.setQuaternion(0.0,0.79 ,0.0055 ,0.61);
      //n.normalize();
```

```
      cout <<m<<endl;
      cout <<"K = "<<kValue<<endl;
      cout<<"initial "<<qHat<<endl;
      cout <<"Filter : Initialized "<<endl;

}//end Constructor


//---------------------------------------------------------------------
// Function:      ~Filter()
// Return Value:  None
// Parameters:    None
// Description:   Destructor
//---------------------------------------------------------------------
Filter::~Filter()
{
      cout <<"deleting Filter"<<endl;
      delete [] samples;
      delete [] biasError;
      fresults.close();
      mySampler->~Sampler();

}//end destructor


//---------------------------------------------------------------------
// Function:      Filter(Filter &)
// Return Value:  None
// Parameters:    Filter
// Description:   Copy constructor
//---------------------------------------------------------------------
Filter::Filter(const Filter &F)
{
      cerr <<"copy constructor NOT implemented";
}// end copy constructor


//---------------------------------------------------------------------
// Function:      start()
// Return Value:  delta t - double
// Parameters:    None
// Description:   Starts the filtering process,calculates the initial
//                      bias error
//---------------------------------------------------------------------
double Filter::start()
{
      cout << "\nFilter : Filtering started"<<endl;

      mySampler->findZeroVoltages(selfCalibration);

      getNewSample();

      initialBiasError();

      correctBiasError();
```

```
      setElements();
      // sets Earth's magnetic field to the first reading
      n=b;

      calculateError();

      createXTranspose();

      calculateDeltaQ();

      calculateqDot();

      calculateResult();

//    logResults();

      return deltaT;
}//end start()


//-------------------------------------------------------------------
// Function:      estimateRotation()
// Return Value:  qHat - quaternion - estimated result
// Parameters:    None
// Description:   Estimate new rotation, must call itself for
//                        continuous estimation
//-------------------------------------------------------------------
Quaternion Filter::estimateRotation()
{
      getNewSample();

      correctBiasError();

      setElements();

      calculateError();

      createXTranspose();

      calculateDeltaQ();

      calculateqDot();

      calculateResult();

//    logResults();

      return qHat;
}// end estimateRotation()
```

```
//---------------------------------------------------------------------
// Function:      initialBiasError()
// Return Value:  None
// Parameters:    None
// Description:   Calculates initial bias errors and sets them
//---------------------------------------------------------------------
void Filter::initialBiasError()
{
      int biasNumber = tau/10;

      // check for alpha
      if (((deltaT * kValue) < 1.0) && ((deltaT * kValue) > 0.0)){
            cerr << "Alpha is "<<(deltaT*kValue)<<endl;
      }else {
            cerr <<"WARNING Filter : Alpha is "<< (deltaT*kValue)
                  << "deltaT "<<deltaT<<" K " <<kValue<<endl;
      }
      for (int i=0;i<biasNumber;i++){
            biasError[0] += samples[0]/biasNumber;  //angular rate p
            biasError[1] += samples[1]/biasNumber;  //angular rate q
            biasError[2] += samples[2]/biasNumber;  //angular rate r
      }
      biasError[0] = -biasError[0];
      biasError[1] = -biasError[1];
      biasError[2] = -biasError[2];

      return;
}// end initialBiasError()


//---------------------------------------------------------------------
// Function:      correctBiasError()
// Return Value:  None
// Parameters:    None
// Description:   For every reading corrects angular rates with the
//                        initial bias error
//---------------------------------------------------------------------
void Filter::correctBiasError()
{
      sampleWeight = deltaT/tau;
      biasWeight = 1-sampleWeight;

      for (int i=0;i<3;i++){
            samples[i] += (biasWeight * biasError[i])-(sampleWeight
                                                * samples [i]);
      }

      return;
}// end correctBiasError()
```

```
//---------------------------------------------------------------------
// Function:      getNewSample()
// Return Value:  None
// Parameters:    None
// Description:   Calls Sampler to get new averaged reading
//---------------------------------------------------------------------
void Filter::getNewSample()
{
      mySampler->getData(samples, deltaT);
/*
      for (int k=0;k<9;k++){
            if (k==0) cerr<<"\nBody Rates      [ ";
            if (k==3) cerr<<"Accelerometer   [ ";
            if (k==6) cerr<<"Magnetometer    [ ";
            cerr<<samples[k]<<" ";
            if ((k==2) || (k==5)){
                  cerr<<"]  "<<endl;
            }

      }
      cerr<<"]"<<endl;
*/
      return;
}// end getNewSample()


//---------------------------------------------------------------------
// Function:      setElements()
// Return Value:  None
// Parameters:    None
// Description:   set b,h,bw with new readings
//---------------------------------------------------------------------
void Filter::setElements()
{

      //reset angular rates
      bw.setQuaternion(0.0, samples[0], samples[1], samples[2]);

      //reset accelerometer measurements
      h.setQuaternion(0.0,samples[3],samples[4],samples[5]);
      h.normalize();

      //reset magnetometer measurements
      b.setQuaternion(0.0,samples[6],samples[7],samples[8]);
      b.normalize();

      return;
}// end setElements()
```

```
//---------------------------------------------------------------------
// Function:      calculateError()
// Return Value:  None
// Parameters:    None
// Description:   Calculates error vector
//---------------------------------------------------------------------
void Filter::calculateError()
{
      static Quaternion errH("error h");
      static Quaternion errB("error b");
      hHat = qHat.toBody(m);
      bHat = qHat.toBody(n);
      errH = hHat - h;
      errB = bHat - b;
      errorMagnetitude = errH.dotProduct(errH) + errB.dotProduct(errB);
      errorqHat.insertColQuaternion ( (hHat-h) , (bHat-b) , 0);

      return;
}// end calculateError()


//---------------------------------------------------------------------
// Function:      createXTranspose()
// Return Value:  None
// Parameters:    None
// Description:   Creates X Transpose matrix
//---------------------------------------------------------------------
void Filter::createXTranspose()
{
      xTranspose.insertRowQuaternion(m.partialDerivative(qHat,QHATZERO,
            QHATZERO),n.partialDerivative(qHat,QHATZERO,QHATZERO) , 0);
      xTranspose.insertRowQuaternion(m.partialDerivative(qHat,QHATONE,
            ONEINV) ,n.partialDerivative(qHat,QHATONE,ONEINV) , 1);
      xTranspose.insertRowQuaternion(m.partialDerivative(qHat,QHATTWO,
            TWOINV) , n.partialDerivative(qHat,QHATTWO,TWOINV) , 2);
      xTranspose.insertRowQuaternion(m.partialDerivative(qHat,QHATTHREE
      ,THREEINV) ,n.partialDerivative(qHat,QHATTHREE,THREEINV) , 3);
      return;
}// end createXTranspose()

//---------------------------------------------------------------------
// Function:      calculateDeltaQ()
// Return Value:  None
// Parameters:    None
// Description:   Calculates delta q in matrix form
//---------------------------------------------------------------------
void Filter::calculateDeltaQ()
{
      xTranspose.transpose(x);
      temp = (xTranspose * x);
      temp = temp.invert();
      temp1 = temp * (xTranspose*errorqHat);
      convertToDeltaq(temp1);
      return;
}// end calculateDeltaQ()
```

```
//-------------------------------------------------------------------
// Function:      convertToDeltaQ()
// Return Value:  None
// Parameters:    MAT - calculated delta q matrix
// Description:   Converts deltaQ 4x1 in to quaternion
//-------------------------------------------------------------------
void Filter::convertToDeltaq(Matrix & MAT)
{
      if ((MAT.getRow()==4) && (MAT.getCol()==1)){

      deltaq.setQuaternion(MAT.getElement(0,0),MAT.getElement(1,0),

      MAT.getElement(2,0),MAT.getElement(3,0));
            deltaq = (-1 * kValue)*deltaq;
      } else {
            cerr <<"Error in Filter: matrix can not"
                  <<" be converted into quaternion"<<endl;
      }

      return;
}// end convertToDeltaQ()


//-------------------------------------------------------------------
// Function:      calculateqDot()
// Return Value:  None
// Parameters:    None
// Description:   Calculates qDot quaternion
//-------------------------------------------------------------------
void Filter::calculateqDot()
{
      qDot = 0.5 *(qHat * bw);
      return;
}// end calculateqDot()


//-------------------------------------------------------------------
// Function:      calculateResult()
// Return Value:  None
// Parameters:    None
// Description:   Finds the estimated result
//-------------------------------------------------------------------
void Filter::calculateResult()
{
      qHat = qHat + ((qDot * deltaT) + (deltaT * deltaq));
      qHat.normalize();

      return;
}// end calculateResult()
```

```
//-------------------------------------------------------------------
// Function:      calculateResult()
// Return Value:  None
// Parameters:    None
// Description:   Finds the estimated result
//-------------------------------------------------------------------
void Filter::logResults()
{
      static int steps = 1;

      fresults<<steps<<" "<<qHat<<" Error = "<<errorMagnetitude<<endl;
      steps++;

      return;
}

//-------------------------------------------------------------------
// Function:      (friend) operator<<()
// Return Value:  ostream object
// Parameters:    output - ostream object
//                q - Filter to print
// Description:   Prints the Filter in a form, should be written out
//                of class
//-------------------------------------------------------------------
ostream &operator<<(ostream &output, const Filter &q)
{
      output <<'['<<"Filter's members"<<']'<<q.n<<q.m<<q.h<<q.b<<
            q.bw<<q.deltaq<<q.qDot<<q.hHat<<
            q.bHat<<q.qHat<<q.x<<q.xTranspose<<
            q.errorqHat<<q.errorTranspose<<endl;;

      return output;
}// end operator<<

//end of file Filter.cpp
```

E. CONVERTER.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Converter.h
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Converter
// - Receives data from sensors
// - Updates the double buffer
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __CONVERTER_H__
#define __CONVERTER_H__

#include <iostream.h>
#include <iomanip.h>

#define M_PI        3.14159265358979323846
#define GRAVITY         32.2185  //feet per second


class Converter{

      // overloaded operator<<
      friend ostream &operator<<(ostream &,const Converter &);

public:

      // default constructor
      Converter (int deviceNum, unsigned int rows=300);

      // destructor
      ~Converter();

      // starts the converter
      void start();
```

```
        // checks if half buffer is ready
        bool isHalfReady();

        // transfers half of the double buffer
        void transferData(short *);

        // stops the converter
        void stop();

        // passes the configuration data
        void getConfigData(unsigned short &,unsigned short &,int &,int &,
              double &);

private:

        long bufferSize,        // buffer size
              timeOut;          // time out

        int deviceNumber;       // device number

        short * channelVector,  // channels
              * gainVector,     // gains for channels
              * doubleBuffer;   // double buffer

        double rate;            // scan rate

        short status,           // returned status of converter functions
              numberOfChannels, // number of channels
              halfReady,        // half ready
              daqStop,          // DAQ stop
              sampleT_Base,     // sample time base
              scanT_Base,       // scan time base
              units,            // used units during scanning
              dBEnable,         // flag, double buffer enable
              inputMode,        // converter input mode
              inputRange,       // input range
              polarity,         // input polarity
              drive,            // specifies drive
              gain;             // gain for scanning

        // data points to tranfer to transfer buffer
        unsigned long pointsTransfer;

        unsigned short sampleInterval,      // sample interval
              scanInterval,                 // scan interval
              maxRange;                     // max reading range

        bool ready;                   // flag to indicate half ready

        // initialize converter
        void initConverter();

        // calibrates converter
        void calibrate();
```

```
        // sets events
        void setEvents();

        // restarts the converter
        void reStart();

        // prints the initial values
        void printInitialValues();
};

#endif

// end of file Converter.h
```

F. CONVERTER.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Converter.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Converter class
// - Receives data from sensors
// - Updates the double buffer with this data
// - Transmits the half buffer
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <assert.h>
#include <iomanip.h>
#include "converter.h"
#include "nidaq.h"
#include "nidaqcns.h"
#include "util.h"


//---------------------------------------------------------------------
// Function:      Converter()
// Return Value:  None
// Parameters:    None
// Description:   Default constructor
//---------------------------------------------------------------------
Converter::Converter(int deviceNo, unsigned int rowNo)
: numberOfChannels(9),  // to scan
timeOut (180),          // time out value
status(0),              // initial status
sampleT_Base(-3), // board resolution,will be overwritten by DAQ_Rate
scanT_Base(-3),         // 20 KHz
sampleInterval(100),    // will be overwritten by DAQ_Rate
scanInterval(0),        // no time interval between conversions
dBEnable(1),            // enable double bufferring
halfReady(0),           // check for half ready
```

```
daqStop(0),             // check if DAQ stopped
rate (20000),           // DAQ rate
inputMode(1),           // 0 - DIFF, 1 - RSE, 2 - NRSE
inputRange(0) ,         // (ignored)
polarity(0),            // 0 - -10/+10    1- 0/+10
drive(1),               // drive AI sense to onboard ground
gain(2),                // channels have the same gain
ready (false)           // flag for half ready
{
      if ((rowNo > 0) && (deviceNo > 0)){
            cerr<<" Please wait while calibrating the converter
            #"<<deviceNo<<endl;
            deviceNumber = deviceNo;      // device number
            // units used to express rate  0 - pts/sec  ;  1 - sec/pts
            units=0;
            // the size of the double buffer
            bufferSize = rowNo * numberOfChannels * 2;

            // points to transfer to transfer buffer
            pointsTransfer = bufferSize / 2;

            // channels to scan
            channelVector = new short [numberOfChannels];
            assert (channelVector != 0);

            // gains per channel
            gainVector = new short [numberOfChannels];
            assert (gainVector != 0);

            // double buffer, cirricular buffer
            doubleBuffer = new short [bufferSize];
            assert (doubleBuffer != 0);

            for (int i=0;i<numberOfChannels;i++){
                  gainVector[i]=1;
            }
            switch (polarity){
                  case 0:
                        maxRange = 32767;
                  break;
                  case 1:
                        maxRange = 32767 * 2 + 1;
                  break;
                  default:
                        cerr<<"ERROR : Unknown polarity\n";
                  break;
            }
            calibrate();
            initConverter();
            printInitialValues();
      }else{
            cerr << "Converter can not initialized, "
                  << "buffer row no must be greater than 0"<<endl;
      }
}//end Constructor
```

```
//-----------------------------------------------------------------------
// Function:      ~Converter()
// Return Value:  None
// Parameters:    None
// Description:   Destructor
//-----------------------------------------------------------------------
Converter::~Converter()
{
      stop();
      delete [] doubleBuffer;
      delete [] channelVector;
      delete [] gainVector;
      cout <<"Converter deleted"<<endl;
}// end destructor


//-----------------------------------------------------------------------
// Function:      initConverter()
// Return Value:  None
// Parameters:    None
// Description:   Initialize the converter
//-----------------------------------------------------------------------
void Converter::initConverter()
{
      // init channels with physical numbers on the board
      channelVector[0]=0;
      channelVector[1]=1;
      channelVector[2]=2;
      channelVector[3]=3;
      channelVector[4]=4;
      channelVector[5]=5;
      channelVector[6]=6;
      channelVector[7]=7;
      channelVector[8]=8;

      // init gains for channels
      for (int i=0;i<numberOfChannels;i++){
            gainVector[i]=gain;
      }

      // Configure converter to software triggered and to
      // use on board clock
      status = DAQ_Config (deviceNumber , 0, 0);
      if (status == 0){
            cerr <<"Converter configured"<<endl;
      } else {
            cerr <<"Error : Configure\n";
      }

      // set AI values, they could be changed by
      // the config utility software
      status = AI_Configure (deviceNumber, -1, inputMode, inputRange,
                             polarity, drive);
      if (status == 0){
            cerr <<"AI configured"<<endl;
```

```
	} else {
		cerr <<"Error : AI_configure\n";
	}

	// Configure the time out
	status = Timeout_Config(deviceNumber, timeOut);
	if (status == 0){
		cerr <<"time out set"<<endl;
	} else {
		cerr <<"Error : Time out\n";
	}

	// Enable double buffering operations
	status = DAQ_DB_Config(deviceNumber, dBEnable);
	if (status == 0){
		cerr <<"AD Converter initialized"<<endl;
	} else {
		cerr <<status<< " Problem with initialization\n";
	}
	cout <<"Converter : Initialized\n";
	return ;
}//end initConverter()


//---------------------------------------------------------------------
// Function:	calibrate()
// Return Value:  None
// Parameters:	None
// Description:	Calibrates the converter
//---------------------------------------------------------------------
void Converter::calibrate()
{
	// calibrates the converter
	status = Calibrate_E_Series(deviceNumber, ND_SELF_CALIBRATE,
					ND_USER_EEPROM_AREA, 0.0);
	if (status != 0){
		cout <<"Error Calibrate\n";
		cout <<status;
	} else{
		cout <<"\nConverter : AD Converter calibrated";
	}

	// calculates and sets the rate
	status = DAQ_Rate(rate, units, &sampleT_Base, &sampleInterval);
	if (status != 0){
		cerr <<"Error DAQ_Rate\n";
		cerr <<status;
	}else {
		cerr <<"\nRate calculated \n";
	}

	status =0;
	return ;
}//end calibrate()
```

```
//---------------------------------------------------------------------
// Function:      printInitialValues()
// Return Value:  None
// Parameters:    None
// Description:   Prints the initial values
//---------------------------------------------------------------------
void Converter::printInitialValues()
{
      cerr<<"Channel's gain is     : "<<gain<<endl;
      switch (inputMode){
            case 0:
                  cerr<<"Mode         : Differential\n";
            break;
            case 1:
                  cerr<<"Mode         : Referenced Single Ended\n";
            break;
            case 2:
                  cerr<<"Mode         : Non Referenced Single Ended\n";
            break;
            default:
                  cerr<<"Unknown Mode\n";
            break;
      }
      switch (polarity){
            case 0:
                  cerr <<"Polarity             : Bipolar -10/+10\n";
            break;
            case 1:
                  cerr<<"Polarity              : Unipolar 0/+10\n";
            break;
            default:
                  cerr<<"Unknown polarity\n";
            break;
      }
      cerr <<"Device number         : "<<deviceNumber
            <<"\nNumber of Channels    : "<<numberOfChannels
   <<"\nTransfer buffer depth : "<<(bufferSize/(2* numberOfChannels))
      <<"\nDouble buffer depth   : " <<(bufferSize/numberOfChannels)
            <<"\nDouble buffer size    : "<< bufferSize
            <<"\nSample Time Base      : "<<sampleT_Base
            <<"\nSample Interval       : "<<sampleInterval
            <<"\nSampling rate         : "<<rate/numberOfChannels
            <<"\nTime out              : "<<timeOut<<endl;

      return;
} // end printInitialValues()
```

```
//---------------------------------------------------------------------
// Function:      getConfigData()
// Return Value:  None
// Parameters:    None
// Description:   Pass the values to the sampler
//---------------------------------------------------------------------
void Converter::getConfigData(unsigned short &range,
                        unsigned short &myGain, int &halfDepth,int
&block, double & deltaTime)
{
      double tBase = 0.0,nano = 1/1000000000.0, micro = 1/1000000.0;
      range = maxRange;
      myGain = gain;
      halfDepth = (bufferSize / (numberOfChannels*2));
      block = numberOfChannels;
      if (sampleT_Base == -3){
            tBase = 50 * nano;
      }else if (sampleT_Base == -1){
            tBase = 200 * nano;
      }else if (sampleT_Base == 1){
            tBase = micro;
      }else if (sampleT_Base == 2){
            tBase = 10 * micro;
      }else if (sampleT_Base == 3){
            tBase = 100 * micro;
      }else if (sampleT_Base == 4){
            tBase = 0.001;
      }else if (sampleT_Base == 5){
            tBase = 0.01;
      }else {
            cerr << "ERROR Converter = Unknown sample time base\n";
            tBase = 0.0;
      }//   deltaTime = 9 * halfDepth * tBase;
      deltaTime = 9 * halfDepth / rate;
      cerr<<"Update rate : "<< deltaTime<<" seconds\n";
      return;
} // end getConfigData()

//---------------------------------------------------------------------
// Function:      start()
// Return Value:  None
// Parameters:    None
// Description:   Starts the data acquisition process
//---------------------------------------------------------------------
void Converter::start()
{
      // set the scanning values
      status = SCAN_Setup (deviceNumber, numberOfChannels ,
                           channelVector, gainVector);
      if (status != 0){
            cerr <<status<<" Error SCAN_Setup\n";
      }
      // start the scanning process
      status = SCAN_Start (deviceNumber , doubleBuffer, bufferSize,
            sampleT_Base, sampleInterval, scanT_Base, scanInterval);
```

```
      if (status!=0){
            cout <<status<<"Error SCAN_Start\n";
      }
      status = 0;
      cerr <<"\nConverter : started\n";
      return ;
}//end start()


//---------------------------------------------------------------------
// Function:      stop()
// Return Value:  None
// Parameters:    None
// Description:   Stops the converter
//---------------------------------------------------------------------
void Converter::stop()
{
      status = DAQ_Clear(deviceNumber);
      status = DAQ_DB_Config(deviceNumber , 0);
      status = Timeout_Config(deviceNumber, -1);
      status = 0;
      cout <<"Converter : Stoped and cleared"<<endl;
      return ;
}//end stop()

//---------------------------------------------------------------------
// Function:      setEvents()
// Return Value:  None
// Parameters:    None
// Description:   Sets the events for converter
//---------------------------------------------------------------------
void Converter::setEvents()
{     // no events set
      return ;
}//end setEvents()

//---------------------------------------------------------------------
// Function:      isHalfReady()
// Return Value:  true if half ready
// Parameters:    None
// Description:   First step to receive data from A/D converter
//---------------------------------------------------------------------
bool Converter::isHalfReady()
{
      ready = false;
      // check status
      halfReady =0;
      while (!((halfReady == 1) && (status == 0))){
            // check for half ready
      status = DAQ_DB_HalfReady(deviceNumber, &halfReady, &daqStop);
            if (status != 0){
                  cerr<<"Half ready error"<<endl;
                  reStart();
            }
      }
```

```
      ready = true;
      return ready;
}//end isHalfready()


//---------------------------------------------------------------------
// Function:      transferData()
// Return Value:  None
// Parameters:    None
// Description:   Transfers the half buffer to transfer buffer
//---------------------------------------------------------------------
void Converter::transferData(short * trBuf)
{
      // transfer data
      status = DAQ_DB_Transfer(deviceNumber,trBuf,
                                 &pointsTransfer,&daqStop);
      if (status != 0){
            cerr<<"Transfer error"<<endl;
            reStart();
      }
      return ;
}//end transferData()


//---------------------------------------------------------------------
// Function:      reStart()
// Return Value:  None
// Parameters:    None
// Description:   Restarts the converter
//---------------------------------------------------------------------
void Converter::reStart()
{
      cerr<<" RESTARTING Converter error status : "<<status<<endl;
      status = DAQ_Clear(deviceNumber);
      start();
      return;
} // end restart()


//---------------------------------------------------------------------
// Function:      (friend) operator<<()
// Return Value:  ostream object
// Parameters:    output - ostream object
//                C - Converter to print
// Description:   Prints the Converter information
//---------------------------------------------------------------------
ostream &operator<<(ostream &output, const Converter &C)
{
      output      <<"\nConverter : depth= "<<C.bufferSize
                  <<endl;

      return output;
}// end operator<<
//end of file Converter.cpp
```

G. SAMPLER.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Sampler.h
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Sampler
// - Receives data from A/D converter
// - Averages, scales and converts this data
// - Updates measurements with these data
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __SAMPLER_H__
#define __SAMPLER_H__

#include <iostream.h>
#include <iomanip.h>
#include "converter.h"

#define M_PI        3.14159265358979323846
#define GRAVITY         32.2185  //feet per second


class Sampler{

      // overloaded operator<<
      friend ostream &operator<<(ostream &,const Sampler &);

public:

      // default constructor
      Sampler (Converter *);

      // destructor
      ~Sampler();
```

```
      // finds zero level voltages for calibration
      void findZeroVoltages(bool);

      // get data from AD converter
      void getData(double *, double &);

private:

      //AD converter
      Converter * myAD;

      int bufferDepth,        // transfer buffer depth
            blockSize,        // block, row size
            bufferSize,       // transfer buffer size
            count;            // number of readings

      unsigned short maxReading,    // A/D converter max reading
            adGain;                 // A/D converter channel gain

      // variables used for conversions
      double angXConversion ,angZConversion ,angYConversion ,
            accXConversion,accYConversion,accZConversion,
            magXConversion,magYConversion,magZConversion,
            angMultiplier,accMultiplier,magMultiplier,
            sampleTime,       // sample time

            // zero level voltage values
            angXV, angYV, angZV, accXV,
            accYV,accZV, magXV, magYV, magZV,

            // scalar numbers for calibration
            angXScalar, angYScalar, angZScalar, accXScalar, accYScalar,
            accZScalar, magXScalar, magYScalar, magZScalar,

            // voltage range for converter
            voltRange;

      short * transferBuffer;     // transfer buffer

      // reads transfer buffer, updates readings
      void readAndUpdate(double *);

      // prints the transfer buffer
      void printTransferBuffer();

      // converts readings
      double convert(double , const int);

      // helper function of convert()
      double convertHelper (double , double , double);

};

#endif
// end of file Sampler.h
```

H. SAMPLER.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Sampler.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Sampler
// - Receives data from A/D converter
// - Averages, scales and converts this data
// - Updates measurements with these data
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <assert.h>
#include <iomanip.h>
#include <fstream.h>
#include <sys/timeb.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sampler.h"
#include "util.h"
#include "converter.h"


//---------------------------------------------------------------------
// Function:      Sampler()
// Return Value:  None
// Parameters:    None
// Description:   Default constructor
//---------------------------------------------------------------------
Sampler::Sampler (Converter * con)
:myAD(con),       // my AD Converter
count(0),         // number of samples
maxReading(0),    // max converter reading
adGain(0),        // channel gain
voltRange(10.0), // converter volt range
```

```
angXV(2.573),angYV(1.65),angZV(2.21), // assign zero voltage levels
accXV(2.515),accYV(2.45),accZV(2.5),  // also will be calculated
magXV(2.45),magYV(2.51),magZV(2.509),  // by findZeroVoltages()
angXScalar(1.0),angYScalar(1.0),angZScalar(1.0), // scalar numbers
accXScalar(2.99),accYScalar(2.0),accZScalar(1.0), // for calibration
magXScalar(1.5),magYScalar(1.8),magZScalar(1.8)
// 2.99 2.99 2.99 2.0 2.0 1.415 4.0 4.0 4.0 tested scalars
{
      myAD->getConfigData(maxReading, adGain, bufferDepth, blockSize,
            sampleTime);

      bufferSize = bufferDepth * blockSize;
      transferBuffer = new short [bufferSize];
      assert (transferBuffer != 0);
      for (int k=0;k<bufferSize;k++){
            transferBuffer[k] = 0;
      }
      angMultiplier = (90.0*(M_PI/180.0)) ;
      accMultiplier = (4 * GRAVITY);
      magMultiplier = 2.0 ;

}//end Constructor


//---------------------------------------------------------------------
// Function:      ~Sampler()
// Return Value:  None
// Parameters:    None
// Description:   Destructor
//---------------------------------------------------------------------
Sampler::~Sampler()
{
      cerr<<"deleting sampler"<<endl;
      delete [] transferBuffer;
}// end destructor


//---------------------------------------------------------------------
// Function:      findZeroVoltages()
// Return Value:  None
// Parameters:    selfCalibrate - bool - to make self calibration
// Description:   Finds the zero level voltages, gets data from
//                converter as many as repeat number
//---------------------------------------------------------------------
void Sampler::findZeroVoltages(bool selfCalibrate)
{
      if (selfCalibrate){
            const int REPEAT = 5;
            double values[9]={0.0}, zeros[REPEAT][6], total=0.0;
            long sum = 0;

            for (int r=0;r<REPEAT;r++){

                  // check if half buffer is ready
                  while (!myAD->isHalfReady()){;}
```

```
            //call transfer function of AD
            myAD->transferData(transferBuffer);

            // find the averaged values
            for (int i=0;i<blockSize;i++){
                  for (int j=0;j<bufferSize; j+=blockSize){
                        sum += transferBuffer[j+i];
                  }
                  values[i] = (sum/(bufferDepth*1.0));
                  sum = 0;
            }

            // fill zero level voltages matrix
            for (int t=0;t<6;t++){
                  zeros[r][t]= voltRange * values[t] /
                                    (maxReading * adGain);
            }
            zeros[r][5] *= (accMultiplier /
                              (accMultiplier + GRAVITY));
      }

      // find averaged zero level voltages
      for (int k=0;k<6;k++){
            for (int rr=0;rr<REPEAT;rr++){
                  total += zeros[rr][k];
            }
            zeros[0][k] = total/REPEAT;
            total=0.0;
      }

      // assign zero level voltages
      angXV = zeros[0][0];
      angYV = zeros[0][1];
      angZV = zeros[0][2];
      accXV = zeros[0][3];
      accYV = zeros[0][4];
      accZV = zeros[0][5];
/*
      for (int tt=0;tt<6;tt++){
            cerr<<" "<<zeros[0][tt];
      }
*/
      cerr<<"Sampler calculated zero level voltages, by sampling"
            <<REPEAT<<" times\n\n";
} else { // use hard coded values
cerr<<"\nSampler is using hard coded values to calibrate\n";
}

// assign conversion values with zero level voltages
angXConversion = maxReading * angXV * adGain / voltRange;// 2.573
angYConversion = maxReading * angYV * adGain / voltRange; // 1.65
angZConversion = maxReading * angZV * adGain / voltRange; // 2.21

accXConversion = maxReading * accXV * adGain / voltRange; // 2.51
```

```
      accYConversion = maxReading * accYV * adGain / voltRange; // 2.46
      accZConversion = maxReading * accZV * adGain / voltRange; // 2.48
      magXConversion = maxReading * magXV * adGain / voltRange;
      magYConversion = maxReading * magYV * adGain / voltRange;
      magZConversion = maxReading * magZV * adGain / voltRange;

      return;
} // end findZeroVoltages()

//---------------------------------------------------------------------
// Function:      getData()
// Return Value:  None
// Parameters:    double m[] - measurements array from Filter
//                double & - delta t
// Description:   Communicates with converter, receives data into
//                transfer buffer, avarages, scales, converts and
//                updates measurements
//---------------------------------------------------------------------
void Sampler::getData (double * m , double & delta_T)
{

      delta_T = sampleTime;

      //  errors will be taken care of by Converter
      while (!myAD->isHalfReady()){;}

      //call transfer function of AD
      myAD->transferData(transferBuffer);

      //printTransferBuffer();
      readAndUpdate(m);
      return;
}//end getData()


//---------------------------------------------------------------------
// Function:      readAndUpdate()
// Return Value:  None
// Parameters:    double ma[] - same array with the getData()
// Description:   Reads transfer buffer, avarages data and updates
//                measurements
//---------------------------------------------------------------------
void Sampler::readAndUpdate(double * ma)
{
      long sum = 0;

      for (int i=0;i<blockSize;i++){
            for (int j=0;j<bufferSize; j+=blockSize){
                  sum += transferBuffer[j+i];
            }
            ma[i] = convert((sum/(bufferDepth*1.0)) , i);
            sum = 0;
      }
      return;
}// end readAndUpdate()
```

```
//---------------------------------------------------------------------
// Function:      convert()
// Return Value:  converted value - double
// Parameters:    averaged - double - averaged value
//                who -int - related reading no
// Description:   Generic function, converts the averaged value by
//                using related numbers
//---------------------------------------------------------------------
double Sampler::convert(double averaged , const int who)
{
      double result = 0.0;
      switch (who){
            case 0:     // body rate p
                  result = convertHelper(averaged, angXConversion,
                        angMultiplier * angXScalar);
                  break;
            case 1:     // body rate q
                  result = convertHelper(averaged, angYConversion,
                        angMultiplier * angYScalar);
                  break;
            case 2:     // body rate r
                  result = -convertHelper(averaged, angZConversion,
                        angMultiplier * angZScalar);
                  break;
            case 3:     // accelerometer h1
                  result = -convertHelper(averaged, accXConversion ,
                        accMultiplier * accXScalar);
                  break;
            case 4:     // accelerometer h2
                  result = -convertHelper(averaged, accYConversion ,
                        accMultiplier * accYScalar);
                  break;
            case 5:     // accelerometer h3
                  result = -convertHelper(averaged , accZConversion ,
                        accMultiplier * accZScalar);
                  break;
            case 6:     // magnetometer b1
                  result = -convertHelper(averaged , magXConversion,
                        magMultiplier * magXScalar);
                  break;
            case 7:     // magnetometer b2
                  result = -convertHelper(averaged , magYConversion,
                        magMultiplier * magYScalar);
                  break;
            case 8:     // magnetometer b3
                  result = -convertHelper(averaged , magZConversion,
                        magMultiplier * magZScalar);
                  break;
            default:
            cout <<"Error in Sampler: Check your data block size\n";
                  break;
      }
      return result;
}// end convert()
```

```
//---------------------------------------------------------------------
// Function:      convertHelper()
// Return Value:  converted value
// Parameters:    theNumber - double - averaged value
//                adConversion - double - related conversion number
//                mult - double - related multiplier contains scalar
// Description:   Generic function, converts the averaged value by
//                using related numbers
//---------------------------------------------------------------------
double Sampler::convertHelper(double theNumber , double adConversion,
                                            double mult)
{
      return (((theNumber - adConversion) / adConversion) * mult);
}// end convertHelper()


//---------------------------------------------------------------------
// Function:      printTransferBuffer()
// Return Value:  None
// Parameters:    None
// Description:   Prints the transfer buffer
//---------------------------------------------------------------------
void Sampler::printTransferBuffer()
{
      for (int i=0;i<bufferDepth;i++){
            for (int j=0;j<blockSize;j++){
      cout<<setprecision(15)<<transferBuffer[j+(i*blockSize)]<<" ";
            }
            cout <<endl;
      }
      return;
}//end printTransferBuffer()


//---------------------------------------------------------------------
// Function:      (friend) operator<<()
// Return Value:  ostream object
// Parameters:    output - ostream object
//                s - sampler to print
// Description:   Prints the sampler information
//---------------------------------------------------------------------
ostream &operator<<(ostream &output, const Sampler &s)
{
      output      <<"\nsampler : depth= "<<s.bufferDepth
                  <<" block size= "<<s.blockSize
                  <<" transfer buffer size= "<<s.bufferSize
                  <<" current counter= "<<s.count<<endl;

      return output;
}// end operator<<
//end of file Sampler.cpp
```

I. MATRIX.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Matrix.h
// Author : ildeniz DUMAN
// Project Name : Matrix Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Header file for Matrix.cpp
// - Executes matrix operations
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __MATRIX_H__
#define __MATRIX_H__

#include <iostream.h>
#include <iomanip.h>
#include "quaternion.h"


class Matrix{

      // overloaded operator<<
      friend ostream &operator<<(ostream &,const Matrix &);

public:

      // default constructor
      Matrix (char * mname="Matrix",int mrow = 4,int mcol = 4);

      //conversition constructor from a two dimensional double array
      Matrix (char * mname,int arrayRow, int arrayCol,double **);

      // destructor
      ~Matrix();

      // copy constructor
      Matrix (const Matrix &);
```

```
        // matrix invertion
        Matrix invert();

        // transpose
        void transpose (Matrix &) const;

        // Matrix product
        Matrix operator*(const Matrix &) const;

        // Matrix assignment
        Matrix &operator=(const Matrix &);

        // Matrix product and assignment
        Matrix &operator*=(const Matrix &);

        // creates a unit matrix
        Matrix unitMatrix(int);

        // inserta a quaternion in a row
        void insertRowQuaternion(Quaternion & a, Quaternion & b, int);

        // inserta a quaternion in a col
        void insertColQuaternion(Quaternion & d, Quaternion & c, int);

        // return row no
        int getRow(){return row;}

        // return col no
        int getCol(){return col;}

        // returns an element from the matrix
        double getElement(int i, int j){ return matrix[i][j];}

private:

        // the name of the Matrix
        char * name;

        // the elements of a Matrix
        double ** matrix;

        //row and column
        int row, col;

};

#endif

// end of file Matrix.h
```

J. MATRIX.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Matrix.cpp
// Author : ildeniz DUMAN
// Project Name : Matrix Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Matrix
// - Executes matrix operations
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <math.h>
#include <string.h>
#include <assert.h>
#include "Matrix.h"
#include "quaternion.h"


//---------------------------------------------------------------------
// Function:      Matrix()
// Return Value:  None
// Parameters:    mname - name of the matrix
//                mrow - row no
//                mcol - col no
// Description:   Default constructor
//---------------------------------------------------------------------
Matrix::Matrix (char* mname,int mrow, int mcol)
:row(mrow),col(mcol)
{
      int length = strlen(mname);
      name = new char[length+1];
      assert (name != 0);
      strcpy(name,mname);

      matrix = new double *[row];
      assert (matrix !=0);
```

```
      for (int x = 0; x<row; x++){
            matrix[x] = new double [col];
            assert (matrix[x] !=0);
      }
      for (int i = 0;i<row;i++){
            for (int j = 0;j<col;j++){
                  matrix[i][j] = 0.0;
            }
      }
}//end Constructor


//-------------------------------------------------------------------
// Function:      ~Matrix()
// Return Value:  None
// Parameters:    None
// Description:   Destructor
//-------------------------------------------------------------------
Matrix::~Matrix()
{
      delete [] name;
      for (int x=0;x<row;x++){
            delete matrix[x];
      }
      delete [] matrix;
}//end destructor


//-------------------------------------------------------------------
// Function:      Matrix(Matrix &)
// Return Value:  None
// Parameters:    MAT - Matrix to copy
// Description:   Copy constructor
//-------------------------------------------------------------------
Matrix::Matrix(const Matrix &MAT)
{
      int length = strlen(MAT.name);

      name = new char[length+1];
      assert(name != 0);
      strcpy(name,MAT.name);
      matrix = new double *[MAT.row];
      assert (matrix !=0);
      for (int x = 0; x<MAT.row; x++){
            matrix[x] = new double [MAT.col];
            assert (matrix[x] !=0);
      }
      row=MAT.row;
      col=MAT.col;
      for (int i = 0;i<MAT.row;i++){
            for (int j = 0;j<MAT.col;j++){
                  matrix[i][j] = MAT.matrix[i][j];
            }
      }
}// end copy constructor
```

```
//---------------------------------------------------------------------
// Function:      Matrix()
// Return Value:  None
// Parameters:    mname - name of the matrix
//                arow - row no
//                acol - col no
//                double [][] - a - two dimesional double array
// Description:   Constructs a matrix from a two dimensional array
//---------------------------------------------------------------------
Matrix::Matrix (char * mname , int arow, int acol,double ** a)
{
      int length = strlen(mname);

      name = new char[length+1];
      assert (name != 0);
      strcpy(name,mname);
      matrix = new double *[arow];
      assert (matrix !=0);
      for (int x = 0; x<arow; x++){
            matrix[x] = new double [acol];
            assert (matrix[x] !=0);
      }
      row = arow;
      col = acol;
      for (int i = 0;i<row;i++){
            for (int j = 0;j<col;j++){
                  matrix[i][j] = a[i][j];
            }
      }
}// end Matrix()


//---------------------------------------------------------------------
// Function:      operator*()
// Return Value:  Matrix
// Parameters:    MAT - Matrix & - Matrix to multiply
// Description:   Calculates the Matrix product
//---------------------------------------------------------------------
Matrix Matrix::operator*(const Matrix &MAT) const
{
      Matrix dest("Product", row , MAT.col);
      double sum = 0.0f;
      for (int i=0;i<row;i++){
            for (int j=0;j<MAT.col;j++){
                  for (int k=0;k<MAT.row;k++){
                        sum += matrix[i][k] * MAT.matrix[k][j];
                  }
                  dest.matrix[i][j]=sum;
                  sum = 0.0;
            }

      }

      return(dest);
}//end operator*
```

```
//---------------------------------------------------------------------
// Function:      operator=()
// Return Value:  Matrix &
// Parameters:    MAT - Matrix &
// Description:   Assigns the MAT to current object
//---------------------------------------------------------------------
Matrix & Matrix::operator=(const Matrix &MAT)
{
      /*if (&MAT != this){
      delete [] name;
      int length = strlen(MAT.name);
      name = new char[length+1];
      assert(name != 0);
      strcpy(name,MAT.name);
      }
      */
      // I let self assingment
      if ((row!=MAT.row) || (col != MAT.col)){
            cout <<"Error in matrix assignment ";
      } else {
          delete [] name;
            int length = strlen(MAT.name);
            name = new char[length+1];
            assert(name != 0);
            strcpy(name,MAT.name);

            for (int i = 0;i<MAT.row;i++){
                  for (int j = 0;j<MAT.col;j++){
                        matrix[i][j] = MAT.matrix[i][j];
                  }
            }
      }
      return (*this);
}//end operator=

//---------------------------------------------------------------------
// Function:      unitMatrix()
// Return Value:  Matrix
// Parameters:    rowOrCol - int - square matrix index
// Description:   Creates a unit matrix
//---------------------------------------------------------------------
Matrix Matrix::unitMatrix (int rowOrCol)
{
      Matrix Unit("unit", rowOrCol , rowOrCol);
      for (int i = 0 ; i<Unit.row ; i++){
            Unit.matrix[i][i] = 1.0;
      }
      return (Unit);

}// end unitMatrix()
```

```
//---------------------------------------------------------------------
// Function:      invert()
// Return Value:  Matrix
// Parameters:    None
// Description:   Calculates the matrix inversion
//---------------------------------------------------------------------
Matrix Matrix::invert()
{
      double multiplier=0.0 , divider =0.0;

      Matrix myUnit=myUnit.unitMatrix(row);

      // square matrix check
      if (row == col){

            //inverting the matrix
            for (int j=0; j<col;j++){
                  for (int i=0; i<row;i++){
                        if (i != j ){
                              multiplier = -matrix[i][j]/matrix[j][j];
                              for (int k=0;k<col;k++){
                                    matrix[i][k] += (multiplier *
                                    matrix [j][k]);
                                    myUnit.matrix[i][k] += (multiplier
                                    * myUnit.matrix[j][k]);
                              }
                        }
                  }
            }

            // final division to make our matrix a unit matrix
            for (int i = 0 ; i<row ; i++){
                  divider = matrix[i][i];
                  if (divider != 0.0){
                        matrix [i][i] /= matrix [i][i];
                        for (int k=0;k<myUnit.row;k++){
                              myUnit.matrix [i][k] /= divider;
                        }
                  }
            }
      } else {
            cout << "Error : Matrix must be a square matrix "<<endl;
      }
      return (myUnit);
}// end unitMatrix()
```

```
//---------------------------------------------------------------------
// Function:      operator*=()
// Return Value:  Matrix
// Parameters:    MAT - Matrix &
// Description:   Calculates the product and assigns the result to
//                current object
//---------------------------------------------------------------------
Matrix & Matrix::operator*=(const Matrix &MAT)
{
      *this = *this * MAT;
      return (*this);
}// end operator*=


//---------------------------------------------------------------------
// Function:      transpose()
// Return Value:  None
// Parameters:    tr - Matrix
// Description:   Finds the transpose of a matrix
//---------------------------------------------------------------------
void Matrix::transpose(Matrix & tr) const
{
      if ((row == tr.col) && (col == tr.row)){
            for (int i=0;i<row;i++){
                  for (int j=0;j<col;j++){
                        tr.matrix[j][i] = matrix[i][j];
                  }
            }
      }
      return;
}// end transpose()


//---------------------------------------------------------------------
// Function:      insertRowQuaternion()
// Return Value:  None
// Parameters:    Q1 , Q2 - Quaternions
//                rowNo - int - row no to insert
// Description:   Takes two quaternions and inserts them in a row by
//                ignoring the w values of the quaternions
//---------------------------------------------------------------------
void Matrix::insertRowQuaternion (Quaternion & Q1, Quaternion  & Q2,
                                                      int rowNo)
{
      if ((row >= rowNo) && (col==6)){
            matrix[rowNo][0] = Q1.getX();
            matrix[rowNo][1] = Q1.getY();
            matrix[rowNo][2] = Q1.getZ();
            matrix[rowNo][3] = Q2.getX();
            matrix[rowNo][4] = Q2.getY();
            matrix[rowNo][5] = Q2.getZ();
      }
      return;
}// end insertRowQuaternion()
```

```
//---------------------------------------------------------------------
// Function:      insertColQuaternion()
// Return Value:  None
// Parameters:    Q1 , Q2 - Quaternions
//                colNo - int - coloumn no to insert
// Description:   Takes two quaternions and inserts them in a coloumn
//                by ignoring the w values of the quaternions
//---------------------------------------------------------------------
void Matrix::insertColQuaternion ( Quaternion  & Q1, Quaternion  & Q2,
                                                      int colNo)
{
      if ((col >= colNo) && (row==6)){
            matrix[0][colNo] = Q1.getX();
            matrix[1][colNo] = Q1.getY();
            matrix[2][colNo] = Q1.getZ();
            matrix[3][colNo] = Q2.getX();
            matrix[4][colNo] = Q2.getY();
            matrix[5][colNo] = Q2.getZ();
      }
      return;
}//end insertColQuaternion()


//---------------------------------------------------------------------
// Function:      (friend) operator<<()
// Return Value:  ostream object
// Parameters:    output - ostream object
//                q - Matrix to print
// Description:   Prints the Matrix in a form, should be written out
//                of class
//---------------------------------------------------------------------
ostream &operator<<(ostream &output, const Matrix &q)
{
      output <<'['<<q.name<<']'<<" "<<q.row<<"x"<<q.col<<endl;;

      for (int k=0;k<q.row;k++){
            for (int m=0;m<q.col;m++){
                  output      <<"   "<<q.matrix[k][m];
            }
            output      <<endl;
      }
      return output;
}// end operator<<

//end of file Matrix.cpp
```

K. QUATERNION.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Quaternion.h
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Header file for quaternion.cpp
// - Executes Qauternion operations
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __QUATERNION_H__
#define __QUATERNION_H__

#include <iostream.h>
#include <iomanip.h>

#define M_PI        3.14159265358979323846


class Quaternion{

      // overloaded operator*
      friend Quaternion operator*(const double,Quaternion &);

      // overloaded operator<<
      friend ostream &operator<<(ostream &,const Quaternion &);

public:

      // default constructor
      Quaternion (char * qname="Quaternion",double ww=0.0,
            double xx=0.0,double yy=0.0,double zz=0.0);

      // destructor
      ~Quaternion();
```

```
// copy constructor
Quaternion (const Quaternion &);

// get data members
double getW();
double getX();
double getY();
double getZ();

// quaternion product
Quaternion operator*(const Quaternion &) const;

// overwriting operator*() to make scalar multiplications
Quaternion operator*(const double);

// Quaternion addition
Quaternion operator+(const Quaternion &) const;

// Quaternion substraction
Quaternion operator-(const Quaternion &) const;

// quaternion assignment
Quaternion &operator=(const Quaternion &);

// quaternion product and assignment
Quaternion &operator*=(const Quaternion &);

// quaternion inverse (conjugate)
Quaternion operator-() const;

//sets the quaternion
void setQuaternion(double , double , double , double);

//converts to Euler Angles
Quaternion toEulerAngles();

// rotate a quaternion about a 3D vector (w=0)
Quaternion rotation(const Quaternion &);

// converts to body coordinates
Quaternion toBody(const Quaternion &);

// dot product
double dotProduct(const Quaternion &);

// quaternion to axis angles
Quaternion toAxisAngles();

// rotation angles(w=0) to quaternion
Quaternion toQuaternion();

// normalizes a quaternion
void normalize();
```

```
      // calculates partial derivative
      Quaternion partialDerivative(const Quaternion &,
            const Quaternion &, const Quaternion &) const;

private:

      // the name of the quaternion
      char * name;

      // the elements of a quaternion
      double w,x,y,z;

};

#endif

// end of file Quaternion.h
```

L. QUATERNION.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Quaternion.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for quaternion
// - Executes Qauternion operations
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <math.h>
#include <string.h>
#include <assert.h>
#include "Quaternion.h"


//---------------------------------------------------------------------
// Function:      Quaternion()
// Return Value:  None
// Parameters:    None
// Description:   Default constructor
//---------------------------------------------------------------------
Quaternion::Quaternion (char* qname,double ww,double xx,double yy,
                                    double zz)
:w(ww),x(xx),y(yy),z(zz)
{
      int length = strlen(qname);
      name = new char[length+1];
      assert (name != 0);
      strcpy(name,qname);

}//end Constructor
```

```
//-----------------------------------------------------------------------
// Function:      ~Quaternion()
// Return Value:  None
// Parameters:    None
// Description:   Destructor
//-----------------------------------------------------------------------
Quaternion::~Quaternion()
{
      //cout <<"deleting "<<*this<<endl;
      delete [] name;
}//end destructor


//-----------------------------------------------------------------------
// Function:      Quaternion(Quaternion &)
// Return Value:  None
// Parameters:    Quaternion
// Description:   Copy constructor
//-----------------------------------------------------------------------
Quaternion::Quaternion(const Quaternion &QUAT1)
{
      int length = strlen(QUAT1.name);

      name = new char[length+1];
      assert(name != 0);
      strcpy(name,QUAT1.name);
      w = QUAT1.w;
      x = QUAT1.x;
      y = QUAT1.y;
      z = QUAT1.z;
}// end copy constructor


//-----------------------------------------------------------------------
// Function:      setQuaternion()
// Return Value:  None
// Parameters:    double a,b,c,d - values to set
// Description:   Sets the quaternion with new values
//-----------------------------------------------------------------------
void Quaternion::setQuaternion (double a, double b, double c, double d)
{
      w=a;
      x=b;
      y=c;
      z=d;
      return;
}// end setQuaternion()


// get functions returns the data members
double Quaternion::getW(){return w;}
double Quaternion::getX(){return x;}
double Quaternion::getY(){return y;}
double Quaternion::getZ(){return z;}
```

```
//-------------------------------------------------------------------
// Function:      operator*()
// Return Value:  Quaternion
// Parameters:    QUAT - Quaternion & - First rotation
// Description:   Calculates the quaternion product
//-------------------------------------------------------------------
Quaternion Quaternion::operator*(const Quaternion &QUAT) const
{
      Quaternion dest("Quaternion Product");
      dest.w  =   QUAT.w * w - QUAT.x * x - QUAT.y * y - QUAT.z * z;
      dest.x  =   QUAT.w * x + QUAT.x * w - QUAT.y * z + QUAT.z * y;
      dest.y  =   QUAT.w * y + QUAT.y * w - QUAT.z * x + QUAT.x * z;
      dest.z  =   QUAT.w * z + QUAT.z * w - QUAT.x * y + QUAT.y * x;

      return(dest);
}//end operator*


//-------------------------------------------------------------------
// Function:      operator*(double) overwriten
// Return Value:  Quaternion
// Parameters:    NUM - double - a scalar to multiply
// Description:   Calculates the scalar product, double must be on
//                the right
//-------------------------------------------------------------------
Quaternion Quaternion::operator*(const double NUM)
{
      w=NUM*w;
      x=NUM*x;
      y=NUM*y;
      z=NUM*z;
      return (*this);
}//end operator*(double)


//-------------------------------------------------------------------
// Function:      operator+
// Return Value:  Quaternion
// Parameters:    QUAT - Quaternion - a quaternion to add
// Description:   Calculates the quaternion addition
//-------------------------------------------------------------------
Quaternion Quaternion::operator+(const Quaternion &QUAT) const
{
      Quaternion add("addition");
      add.w = w + QUAT.w;
      add.x = x + QUAT.x;
      add.y = y + QUAT.y;
      add.z = z + QUAT.z;

      return (add);
}// end operator+()
```

```
//-------------------------------------------------------------------
// Function:      operator-
// Return Value:  Quaternion
// Parameters:    QUAT - Quaternion - a quaternion to substract
// Description:   Calculates the quaternion substraction
//-------------------------------------------------------------------
Quaternion Quaternion::operator-(const Quaternion &QUAT) const
{
      Quaternion add("substraction");
      add.w = w - QUAT.w;
      add.x = x - QUAT.x;
      add.y = y - QUAT.y;
      add.z = z - QUAT.z;

      return (add);
}// end operator-()


//-------------------------------------------------------------------
// Function:      operator=()
// Return Value:  Quaternion &
// Parameters:    QUAT2 - Quaternion &
// Description:   Assigns the QUAT2 to current object
//-------------------------------------------------------------------
Quaternion & Quaternion::operator=(const Quaternion &QUAT2)
{
      /*if (&QUAT2 != this){
      delete [] name;
      int length = strlen(QUAT2.name);
      name = new char[length+1];
      assert(name != 0);
      strcpy(name,QUAT2.name);
      }
      I let self assingment      */
      w = QUAT2.w;
      x = QUAT2.x;
      y = QUAT2.y;
      z = QUAT2.z;

      return (*this);
}//end operator=


//-------------------------------------------------------------------
// Function:      operator*=()
// Return Value:  Quaternion
// Parameters:    QUAT3 - Quaternion &
// Description:   Calculates the product and assigns the result to
//                current object
//-------------------------------------------------------------------
Quaternion & Quaternion::operator*=(const Quaternion &QUAT3)
{
      *this = *this * QUAT3;
      return (*this);
}// end operator*=
```

```
//-----------------------------------------------------------------
// Function:      operator-()
// Return Value:  Quaternion
// Parameters:    None
// Description:   Calculates the inverse (conjugate) quaternion
//-----------------------------------------------------------------
Quaternion Quaternion::operator-() const
{
      Quaternion temp("Conjugate");
      temp.w = w;
      temp.x = -x;
      temp.y = -y;
      temp.z = -z;
      return (temp);
}//end operator-


//-----------------------------------------------------------------
// Function:      rotation()
// Return Value:  Quaternion
// Parameters:    QUAT4 - Quaternion & - vector to rotate around
// Description:   Calculates the rotation quaternion
//-----------------------------------------------------------------
Quaternion Quaternion::rotation(const Quaternion &QUAT4)
{
      Quaternion temp("Rotation");
      temp = *this * ( QUAT4 * (-(*this)));
      return (temp);
}//end Rotation()


//-----------------------------------------------------------------
// Function:      toBody()
// Return Value:  Quaternion
// Parameters:    QUAT4 - Quaternion & - vector to convert to body
//                coordinates
// Description:   Converts into the body coordinates
//-----------------------------------------------------------------
Quaternion Quaternion::toBody(const Quaternion &QUAT)
{
      Quaternion temp("to Body");
      temp = ((-(*this)) * (QUAT * (*this)));
      return (temp);
}//end Rotation()
```

```
//-----------------------------------------------------------------
// Function:      dotProduct()
// Return Value:  double
// Parameters:    QUAT4 - Quaternion &
// Description:   Calculates the dot product of quaternions
//-----------------------------------------------------------------
double Quaternion::dotProduct(const Quaternion &QUAT5)
{
      double dotproduct;
      dotproduct = (w * QUAT5.w) + (x * QUAT5.x) + (y * QUAT5.y) +
            (z * QUAT5.z);

      return (dotproduct);
} //end dotProduct()


//-----------------------------------------------------------------
// Function:      normalize()
// Return Value:  None
// Parameters:    None
// Description:   Normalizes the quaternion
//-----------------------------------------------------------------
void Quaternion::normalize()
{
      double normal;
      normal = (x * x) + (y * y) + (z * z) + (w * w);
      x = x / sqrt(normal);
      y = y / sqrt(normal);
      z = z / sqrt(normal);
      w = w / sqrt (normal);
      return;
}//end normalize()


//-----------------------------------------------------------------
// Function:      toAxisAngles()
// Return Value:  Quaternion
// Parameters:    None
// Description:   Calculates the axis angles of quaternion, results
//                must be in glRotatef()
//-----------------------------------------------------------------
Quaternion Quaternion::toAxisAngles()
{
      Quaternion axisAngle("To Axis Angle ");

      double scalarNumber,temp;
      temp = acos(w) * 2.0;
      scalarNumber = sin(temp / 2.0);
      axisAngle.x = x / scalarNumber;
      // aligning the axes
      axisAngle.y = -1 * z / scalarNumber;
      axisAngle.z = y / scalarNumber;
      axisAngle.w = (temp * (360 / 2.0)) / M_PI;
      return (axisAngle);
}//end toAxisAngles()
```

```
//-----------------------------------------------------------------
// Function:      toQuaternion()
// Return Value:  Quaternion
// Parameters:    None
// Description:   Calculates the quaternion value of three rotations
//-----------------------------------------------------------------
Quaternion Quaternion::toQuaternion()
{
      //current object is a 3D vector with rotation angles at x,y,z
      double radX,radY,radZ,tempx,tempy,tempz;
      double cosx,cosy,cosz,sinx,siny,sinz,cosc,coss,sinc,sins;
      Quaternion quat("Euler To Quaternion");

      // convert angles to radians
      radX =  (x * M_PI) / (360.0 / 2.0);
      radY =  (y * M_PI) / (360.0 / 2.0);
      radZ =  (z * M_PI) / (360.0 / 2.0);
      // half angles
      tempx = radX * 0.5;
      tempy = radY * 0.5;
      tempz = radZ * 0.5;
      cosx = cos(tempx);
      cosy = cos(tempy);
      cosz = cos(tempz);
      sinx = sin(tempx);
      siny = sin(tempy);
      sinz = sin(tempz);

      cosc = cosx * cosz;
      coss = cosx * sinz;
      sinc = sinx * cosz;
      sins = sinx * sinz;

      quat.x = (cosy * sinc) - (siny * coss);
      quat.y = (cosy * sins) + (siny * cosc);
      quat.z = (cosy * coss) - (siny * sinc);
      quat.w = (cosy * cosc) + (siny * sins);
      quat.normalize();
      return(quat);
}// end toQuaternion

//-----------------------------------------------------------------
// Function:      (friend) operator*(double,Quaternion &) overwriten
// Return Value:  Quaternion
// Parameters:    NUM - double - a scalar to multiply
// Description:   Calculates the scalar product
//-----------------------------------------------------------------
Quaternion operator*(const double NUM,Quaternion & quat)
{
    quat.w=NUM*quat.w;
    quat.x=NUM*quat.x;
    quat.y=NUM*quat.y;
    quat.z=NUM*quat.z;
    return (quat);
}//end operator*(double,Quaternion &)
```

```
//-------------------------------------------------------------------
// Function:      toEulerAngles()
// Return Value:  Quaternion
// Parameters:    None
// Description:   Converts quaternion into euler angles , this
//                conversion is inherently ill-defined
//-------------------------------------------------------------------
Quaternion Quaternion::toEulerAngles()
{
      Quaternion euler("Quat to euler");
      double sint,cost,sinv,cosv,sinf,cosf,ex,ey,ez;

      sint = (2*w*y)-(2*x*z);
      cost = sqrt(1- pow(sint,2));
      if (cost != 0.0){
            sinv = ((2*y*z)+(2*w*x))/cost;
            cosv = (1-(2*x*x)-(2*y*y))/cost;
            sinf = (1-(2*x*x)-(2*y*y))/cost;
            cosf = (1-(2*y*y)-(2*z*z))/cost;
      }else {
            sinv = (2*w*x)-(2*y*z);
            cosv = 1-(2*x*x)-(2*z*z);
            sinf = 0.0;
            cosf = 1.0;
      }

      ex = atan2(sinv,cosv);
      ey = atan2(sint,cost);
      ez = atan2(sinf,cosf);  //range -pi +pi

      ex *= 180.0/M_PI;
      ey *= 180.0/M_PI;
      ez *= 180.0/M_PI;

      euler.setQuaternion(0.0, ex,ey,ez);
      return euler;
}//end toEulerAngles()


//-------------------------------------------------------------------
// Function:      partialDerivative()
// Return Value:  Quaternion
// Parameters:    M - vector, as in the X matrix
//                QHAT - q-hat
//                DER - partial derivative of quaternion
//                INVDER - inverse partial derivative of quaternion
// Description:   Calculates the partial derivatives for X Matrix,
//                special to filter
//-------------------------------------------------------------------
Quaternion Quaternion::partialDerivative(const Quaternion & QHAT,
            const Quaternion & DER,const Quaternion & INVDER) const
{
      return ((INVDER*((*this) * QHAT))+((-QHAT)*((*this)*DER)));
}// end partialDerivative()
```

```
//-------------------------------------------------------------------
// Function:      (friend) operator<<()
// Return Value:  ostream object
// Parameters:    output - ostream object
//                q - quaternion to print
// Description:   Prints the quaternion in a form, should be written
//                out of class
//-------------------------------------------------------------------
ostream &operator<<(ostream &output, const Quaternion &q)
{
/*    output <<q.name<<" w= "<<q.w
             <<" x= "<<q.x
             <<" y= "<<q.y
             <<" z= "<<q.z;*/
      output << q.w <<" "<<q.x<<" "<<q.y<<" "<<q.z<<" ";
      return output;
}// end operator<<

//end of file Quaternion.cpp
```

M. UTIL.H

```
//*********************************************************************
//*********************************************************************
//
// File   : Util.h
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Header file for Utilities
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************
#ifndef __UTIL_H__
#define __UTIL_H__


#include <iostream.h>
#include <fstream.h>


// opens a file to read
int openFile (ifstream & ,char * );

// opens a file to write
int writeFile (ofstream & ,char * );

#endif

// end of file Util.h
```

N. UTIL.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : Util.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for Utilities
// - Opens a file to write and read
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//*********************************************************************
//*********************************************************************

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include "util.h"


//-------------------------------------------------------------------
// Function:      openFile()
// Return Value:  None
// Parameters:    inDataFile - & ifstream - ifstream object
//                filename - char * - file name to open
// Description:   Opens a file to read
//-------------------------------------------------------------------
int openFile (ifstream & inDataFile ,char * fileName)
{
      int error = 0;

      inDataFile.open (fileName , ios::in);
      cout <<"Reading "<<fileName<<" for configuration settings\n";

      if (!inDataFile){
            cout <<"File "<<fileName<<" not found\n";
            error = 1;
      }
            return error;
}// end openFile()
```

```
//-------------------------------------------------------------------
// Function:      writeFile()
// Return Value:  None
// Parameters:    inDataFile - & ifstream - ifstream object
//                filename - char * - file name to open
// Description:   Opens a file to write
//-------------------------------------------------------------------
int writeFile(ofstream & outDataFile, char *fileName)
{
      int error = 0;

      outDataFile.open (fileName , ios::out);
      cout <<fileName<<" opened to write\n";

      if (!outDataFile){
            cout <<"File "<<fileName<<" not found\n";
            error = 1;
      }

      return error;
}// end writeFile()

//end of file util.cpp
```

O. TEST.CPP

```
//*********************************************************************
//*********************************************************************
//
// File   : test.cpp
// Author : ildeniz DUMAN
// Project Name : Quaternion Attitude Estimation Filter
// Operating Environment : MS-Windows 95
// Compiler : Visual C++ 5.0
// Date : 01/09/99
//
// Description :
//
// - Source file for test
// - Sample driver file for the filter
//
// Inputs : None
//
// Outputs : None
//
// Process : None
//
// Assumption : None
//
// Warnings during compliation : None
//
// Note : nidaq32.lib, nidaq.h and nidaqcns.h must be added to the
//        project file
//*********************************************************************
//*********************************************************************

#include <math.h>
#include "qaef.h"
#include "quaternion.h"
#include "util.h"


int main(){

      // file name to log data
      char * fname;
      fname = new char[15];

      // delta t
      double dt=0.0;

      // step counter
      int step =1;

      // stream to write file
      ofstream result;
```

```
	// quaternions
	Quaternion drawQuat("d",0.0);
	Quaternion qRotation("q",0.0);

	// filter
	Qaef myQaef(1, 40, 3);

	cerr<<"Enter file name, with .dat : ";
	cin>>fname;
	writeFile(result,fname);

	// start filter and get delta t
	dt = myQaef.start();

	while(true){

		// get estimated result
		drawQuat = myQaef.getResult();

		// convert into angles
		qRotation = drawQuat.toAxisAngles();

		//cerr<<qRotation.getW()<<" ";

		// write results in to file
		result<<(step*dt)<<" "<<qRotation<<endl;

		step++;
	}

	// stop the filter
	myQaef.stop();

	return (0);

} // end main()

// end of file test.cpp
```

LIST OF REFERENCES

[BACH96] Bachmann, E., et. al., "Evaluation of an Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)", *Symposium an Autonomous Underwater Vehicle Technology*, IEEE AUV 96, Monterey, California, June 1996.

[BACH97A] Bachmann, E., *Research Notes: Parameter Optimization Techniques*, Naval Postgraduate School, Monterey, CA, 1997.

[BACH97B] Bachmann, E., *Research Notes: Quaternion Attitude Filter*, Naval Postgraduate School, Monterey, CA, 1997.

[BARA93] Baratoff, G., and Blanksteen, S., *Tracking Devices*, University of Maryland, Washington, 1993.

[BERK97] *Using Quaternions to Represent Rotation*, Berkeley University, CA, www.cs.berkeley.edu/~laura/cs184/quat/quaternion.html

[BIBL95] Bible, S. R., Zyda, M. J., Brutzman, D., "Using Spread-Spectrum Ranging Techniques for Position Tracking in a Virtual Environment", *Proceedings of Network Realities* '95, Boston, Massachusetts, 26 - 28 October 1995,

[BROW97] Brown, R. G., Hwang. P. Y. C., *Introduction to Random Signals and Applied Kalman Filtering*, 2nd Edition, John Wiley, New York, 1997.

[COOK92] Cooke, J. M., Zyda, M. J., Pratt, D. R., McGhee, R. B. "NPSNET: Flight Simulation Dynamic Modeling Using Quaternions", *Presence*, Vol. 1, No. 4, pp. 404-420, Fall 1992.

[CRAI89] Craig, J., *Introduction to Robotics, Mechanics and Control, Second Edition*, Addison-Wesley Publishing Company, Menlo Park, California, 1989.

[CROS98] Crossbow Technology Inc., http://www.xbow.com

[DOUG98] Douglass, B. P., *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison Wesley, Massachusetts, May 1998.

[DURL95] Durlach, N. I., Mavor, A. S., National Research Council, *Virtual Reality: Scientific and Technological Challenges*, National Academy Press, Washington, D.C., 1995.

[FOLE97] Foley, J. D., Van Dam, A., Feiner, S. K. and Hughes J. F., *Computer Graphics Principles and Practice, Second Edition in C*, Addison-Wesley, 1997.

[FOXL94] Foxlin, E., Durlach, N., "An Inertial Head-Orientation Tracker with Automatic Drift Compensation for Use with HMD's", *VRST'94 – Virtual Reality Software and Technology*, Singapore, August 1994.

[FRAN69] Frank, A. A., McGhee, R. B., *Some Considerations Relating to the Design of Autopilots for Legged Vehicles*, *Journal of Terramechanics*, Vol. 6, No. 1, pp. 23-35, Pergamon Press, Great Britain, 1969.

[FREY96] Frey, W., Zyda, M., McGhee, R., Cockayne, B., "Off-the-Shelf Real-Time Human Body Motion Capture for Synthetic Environments", Technical Report NPSCS-96-003, June 1996.

[FUND96] Funda, J., Taylor, R. H., Paul, R. P., "On Homogeneous Transforms, Quaternions, and Computational Efficiency", *IEEE Transactions on Robotics and Automation*, Vol. 6, No 3, June 1990.

[HAND93] Hand, C., "A Survey of 3-D Input Devices", Technical Report CS TR94/2, De Montfort University, Leicester, UK, 1993.

[HENA97] Henault, G. A., *A Computer Simulation Study and Component Evaluation for a Quaternion Filter for Sourceless Tracking of Human Limb Segment Motion*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, September 1996.

[HONE98] Honeywell Solid State Electronics, http://www.ssec.honeywell.com, 1998

[KUO95] Kuo, B. C., *Automatic Control Systems*, 7th Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[MCCA90] McCarthy, J. M., *An Introduction to Theoretical Kinematics*, The MIT Press, Massachusetts, 1990.

[MCGH63] McGhee, R. B., *Identification of Nonlinear Dynamic Systems by Regression Analysis Methods*, Ph.D. Dissertation, Electrical Engineering Department, University of Southern California, June 1963.

[MCGH93] McGhee, R. B., *CS4314 Class Notes: Derivation of Body Angular Rates to Euler Angle Rates Relationship*, Naval Postgraduate School, Monterey, CA, 1993.

[MCGH95] McGhee, R. B., et. al., "An Experimental Study of An Integrated GPS/INS System for Shallow-Water AUV Navigation (SANS)", *Proceedings of the 9th International Symposium on Unmanned Submersible Technology*, Durham, NH, September, 1995, pp. 1-15.

[MCGH96] McGhee, R. B., *Research Notes: A Quaternion Attitude Filter Using Angular Rate Sensors, Accelerometers, and a 3-Axis Magnetometer*, Computer Science Department, Naval Postgraduate School, Monterey, California, 1996.

[MCGH97] McGhee, R. B., *CS4314 Class Notes: Finite and Infinitesimal Quaternion Rotations*, Naval Postgraduate School, Monterey, CA, 1997.

[MCGH98A] McGhee, R. B., *Research Notes: Quaternions Demystified*, Naval Postgraduate School, Monterey, CA, 1998.

[MCGH98B] McGhee, R. B., *Research Notes: Response of Quaternion Filter to Initial Condition Error*, Naval Postgraduate School, Monterey, CA, Sep 08, 1998.

[MCGH98C] McGhee, R. B., *Research Notes: Linearization of Quaternion Attitude Filter*, Naval Postgraduate School, Monterey, CA, Sep 03,1998.

[MCKI98] McKinney Technology, Doug McKinney, 9 Glenn Avenue, Prunedale, CA, 93907.

[MEYE92] Meyer, K., Applwhite, H. L., and Biocca, F. A., "A Survey of Position Trackers", *Presence: Teleoperators and Virtual Environments*, Spring, 1992, Volume 1, Number 2, pp. 173-200.

[NATI98] National Instruments Corporation, http://www.natinst.com, 1998.

[NATI98A] National Instruments Corporation, *NI-DAQ Function Reference Manual for PC Compatibles*, Version 6.1, Part Number 321645C-01, April 1998.

[NATI98B] National Instruments Corporation, *NI-DAQ User Manual for PC Compatibles*, Version 6.1, Part Number 321644C-01, April 1998.

[PERV82] Pervin, E., Webb, J. A., *"Quaternions in Computer Vision and Robotics"*, CMU-CS-82-150, 1982.

[PRAT94] Pratt, D. R., Barham, P. T., Locke, J., Zyda, M. J., Eastman, B., Moore, T., Biggers, K., Douglass, R., Jacobsen, S., Hollick, M., Granieri, J., Ko, H. and Badler, N. I., "Insertion of an Articulated Human into a Networked Virtual Environment", *Proceedings of the Fifth Annual Conference on AI,*

Simulation and Planning in High Autonomy Systems: Distributed Interactive Simulation Environments, IEEE Computer Society Press, Gainesville, Florida, December 7-9, 1994, pp. 84-90.

[ROBE97] Roberts, R. L., *Analysis, Experimental Evaluation, and Software Upgrade for Attitude Estimation by the Shallow-Water AUV Navigation System (SANS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1997.

[RATI98] Rational Software Corporation, *Rational Rose 98 Student Edition*, http://www.rational.com, 1998.

[SHOE85] Shoemake, K., "Animating Rotation with Quaternion Curves", *SIGGRAPH 85*, pp. 245-254, ACM Press, 1985.

[SHOE94] Shoemake, K., *Quaternions* University of Pennsylvania, Philadelphia, PA, 1994.

[SKOP96] Skopowski, P. F., *Immersive Articulation of the Human Upper Body in a Virtual Environment*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, December 1996.

[TOKI98] Tokin America Inc, http://www.tokin.com, 1998.

[USTA99] Usta, U. Y., *Comparison of Quaternion, Euler Angles, and Joint Angle Animation of Human Figure Models*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, March 1999.

[WATT98] Watt, A., Watt, M., *Advanced Animation and Rendering Techniques, Theory and Practice*, ACM Press, Addison-Wesley, New York, New York, 1998.

[YUN97] Yun, X., Bachmann, E. R., McGhee, R. B., Whalen, R. H., Roberts, R. L., Knapp, R. G., Healey, A. J., Zyda, M. J., "Testing and Evaluation of an Integrated GPS/INS System for Small AUV Navigation (SANS)", *Proceedings of the 10th International Symposium on Unmanned Untethered Submersible Technology* (UUST), Durham, NH, September 7-10, 1997.

[ZYDA97] Zyda, M. J., McGhee, R. B., *Inertial Motion Tracking Technology for Inserting Humans Into a Networked Synthetic Environment*, Proposal to Army Research Office (ARO Proposal No. 37660-MA), Naval Postgraduate School, Monterey, California, 1997.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Deniz Kuvvetleri Komutanligi 2
   Personel Tedarik ve Yetistirme Daire Baskanligi
   06100 Bakanliklar, Ankara
   TURKEY

4. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, California 93943-5000

5. Dr. Robert B. McGhee, Professor 2
   Computer Science Department Code CS/MZ
   Naval Postgraduate School
   Monterey, California 93943-5000

6. Eric Bachmann, Instructor 1
   Computer Science Department Code CS/BC
   Naval Postgraduate School
   Monterey, California 93943-5000

7. Ildeniz Duman 2
   Ilica Mah. Zumrut Sok. No:22/3
   35320 Narlidere, Izmir
   TURKEY

8. Orta Dogu Teknik Universitesi 1
   Department of Computer Engineering
   06531 Ankara
   TURKEY

9. Bogazici Universitesi 1
   Department of Computer Engineering
   80815 Bebek, Istanbul
   TURKEY

10. Bilkent Universitesi..... 1
Department of Computer Engineering and Information Science
06533 Bilkent, Ankara
TURKEY